Falsification of Autonomous Systems in Rich Environments

KHEN ELIMELECH, King's College London, UK
MORTEZA LAHIJANIAN, University of Colorado Boulder, USA
LYDIA E. KAVRAKI, Rice University, USA
MOSHE Y. VARDI, Rice University, USA

Validating the behavior of autonomous Cyber-Physical Systems (CPS) and Artificial Intelligence (AI) agents, which rely on automated controllers, is an objective of great importance. In recent years, Neural-Network (NN) controllers have been demonstrating great promise and experiencing tremendous popularity. Unfortunately, such learned controllers are often not certified and can cause the system to suffer from unpredictable or unsafe behavior. To mitigate this issue, a great effort has been dedicated to automated verification of systems. Specifically, works in the category of "black-box testing" rely on repeated system simulations to find a falsifying counterexample—a system run that violates a specification. As running high-fidelity simulations is computationally demanding, the goal of falsification approaches is to minimize the simulation effort needed to return a falsifying example. This often proves to be a great challenge, especially when the tested controller is well-trained. This work contributes a novel falsification approach for autonomous systems under formal specification operating in uncertain environments. We are especially interested in CPS operating in rich, semantically-defined, open environments, which yield high-dimensional, simulation-dependent sensor observations as inputs to the controller. Our approach introduces a novel reformulation of the falsification problem as the problem of planning a trajectory for a "meta-system," which wraps and encapsulates the examined system; we call this approach: meta-planning. This approach results in testing fewer inputs, compared to serial input sampling, while making minimal assumptions on the system, and posing no limitation on the specification, environment, or controller, which is treated as a black-box. It also avoids redundant calculations and requires less effort for each test, by invoking only incremental updates to the autonomous-system's trajectory at each iteration, using partial simulations. This formulation can be solved with standard sampling-based motion-planning techniques (like RRT), can gradually integrate domain knowledge to improve the search, based on its availability, and can even work with no domain knowledge at all. We support these ideas with an experimental study on falsification of an obstacle-avoiding autonomous car with a NN controller, where meta-planning demonstrates superior performance over alternative approaches.

ACM Reference Format:
Khen Elimelech, Morteza Lahijanian, Lydia E. Kavraki, and Moshe Y. Vardi. 2026. Falsification of Autonomous Systems in Rich Environments. *ACM Trans. Cyber-Phys. Syst.* X, X (March 2026), 31 pages. https://doi.org/10.1145/3801740

Work on this paper was supported by Office of Naval Research (ONR) MURI grant no. N00014-20-1-2787.

Authors' Contact Information: Khen Elimelech, khen.elimelech@kcl.ac.uk, King's College London, London, UK; Morteza Lahijanian, morteza.lahijanian@colorado.edu, University of Colorado Boulder, Boulder, CO, USA; Lydia E. Kavraki, kavraki@rice.edu, Rice University, Houston, TX, USA; Moshe Y. Vardi, vardi@rice.edu, Rice University, Houston, TX, USA.

ACM 2378-9638/2026/3-ART
https://doi.org/10.1145/3801740

1 Introduction

1.1 Background and motivation

As autonomous Cyber-Physical Systems (CPS) and Artificial Intelligence (AI) agents become embedded in various aspects of modern life, our ability to certify their behavior and ensure their trustworthiness also becomes increasingly important [1]. To operate autonomously, such systems and agents often rely on automated controllers. These are designed to translate a stream of sensor observations or system states into a stream of commands (controls) to execute, in order to maintain a safe behavior, or robustly perform a specified task. Traditionally, controllers had to be expertly designed, e.g., by meticulously considering the physical and mechanical aspects of the system. In recent years, however, Neural-Network (NN) controllers have been experiencing a surge in popularity. These can handle complex, high-dimensional sensor observations, such as images, and enable effective control of highly-complex dynamical systems, such as race cars, soft robots, high Degree-of-Freedom (DoF) manipulators, and dexterous robot hands—all known as great challenges to the control and robotics communities. Such controllers are typically built ("trained") by compressing numerous interaction examples ("training data") using statistical machine learning techniques, in an attempt to yield a desired behavior. Common techniques include Reinforcement Learning (RL) [2], which rely on repeated trial-and-error control attempts, until apparent convergence, and Imitation Learning [3], which rely on control demonstrations of either a human operator or a traditional controller. Unfortunately, such learning methods generally do not provide a guarantee that the resulting controller is robust and should always exhibit the desired behavior. Hence, reliance on learned controllers can cause the system to suffer from unsafe behavior unpredictably.

1.2 Problem positioning and definition

To mitigate the aforementioned issues, a great effort has been dedicated to automated verification of CPSs [4], which, as we will discuss, has proven to be a formidable challenge. This work comes to contribute to this effort. A system [5], in technical view, is a machine that maps an input signal to an output signal. For system verification, we wish to examine how the variability of the possible inputs may affect the output, which should be tested against a formal specification, indicating a desired behavior or safety requirements. Specifications may be given in various formats, e.g., using automatons, state predicates, or temporal logic [6]. We are especially interested in *autonomous* systems, in which we care to test the automated system controller, for which the inputs are the stream of sensor observations and/or the initial conditions. This comes in contrast to controllable systems, where we want to test the system behavior for different streams of control commands as inputs. We recognize that there has been a considerable effort to develop techniques for robustness verification for NN controllers or AI-based system components in isolation [7–10]; yet, these approaches are inadequate in cases like ours, when the controller inputs and outputs are part of the closed-loop autonomous system, and the specification is defined at the system-level.

Works on automated verification of systems [4] can broadly be divided into two categories. Works in the first category rely on reachability analysis, in order to answer the question "can the system end up in an unsafe state?" Since such analysis requires knowledge on the system dynamics, it is often referred to as "white-box verification." Prominently, many such techniques for verification of traditional continuous controllers rely on Hamilton-Jacobi reachability analysis [11, 12]. While it is more challenging for NN controllers, due to their deep non-linearity and numerous parameters, some recent advances showed that such analysis can be performed for NN controllers with known, exploitable architecture and/or low-dimensional inputs [13–16]. Nevertheless, as the reachability problem is generally undecidable [17], this type of verification is limited to relatively simple systems or systems under specific constraints. Such techniques are also not natively suitable for

temporal specifications, which are defined over state trajectories. Works in the second category rely on performing repeated simulations of a system run, in an attempt to find a counterexample, which would *falsify* the controller, i.e., an example of a system run, under some conditions, that violates the specification. Since this type of analysis typically does not require knowledge on the system dynamics, but only examination of input-output pairs, it is often referred to as "black-box verification" or "black-box testing." Various falsification approaches suggest strategies for effectively choosing promising test cases. As running high-fidelity simulations is often computationally demanding, the goal (and metric of success) of such approaches is to minimize the total simulation effort (length) needed to return a falsifying example. This often proves to be a great challenge, especially when the controller is well-trained.

This paper suggests a novel falsification approach for autonomous systems under formal specification operating in *uncertain environments*. We are especially interested in CPS operating in *rich, semantically-defined, open environments*, which may yield *high-dimensional sensor observations* as inputs to the controller—as is the case in sensor-based autonomous driving. While we are especially interested in systems with NN controllers, this work is, in fact, relevant to verification of general controllers, which are treated as black-boxes.

1.3 Falsification: a review

To properly explain the novelty of our approach in relation to existing solutions, before introducing it, we first provide a summary of the relevant literature. A knowledgeable reader may choose to skip this review directly to Sec. 1.4. Also, while there is some distinction between testing of autonomous and controlled systems, the shared assumption that the input space can be parameterized using a discrete, finite, and samplable set of points makes most falsification techniques applicable to either problem. We will therefore ignore the distinction between the two cases for the rest of this review.

1.3.1 Robustness optimization. These days, the most prominent approach to falsification [18], when the specification is provided as a temporal logic formula over the system trajectory, is reformulating it into an optimization problem of the *signal robustness* [19, 20]—a scalar measure of how "close" the output signal (system trajectory) is to failing the specification. The notion of robustness, often referred to as *quantitative semantics*, is generally limited to continuous signals and temporal logics based on continuous parameters, most prominently Metric Temporal Logic (MTL) [21] and Signal Temporal Logic (STL) [22]. Intuitively, for a specification-satisfying signal, the robustness function returns the radius of the largest cylinder around this signal, in which all contained signals also satisfy the specification; or, in other words, it measures how much we can perturb this signal before the specification is compromised. With that, the falsification problem can be posed as an optimization problem, in which we search for the trajectory of minimal robustness. We note that the robustness function requires domain knowledge and may be non-trivial to calculate in practice.

Since evaluation of the system with a given input requires performing a simulation run, the optimization process is most often based on discrete, serial sampling. The user starts by sampling a random input, to be fed into a simulator, which then generates a system trajectory to be tested (imposing a time limit, if needed). The process then repeats, until a falsifying example is found.

1.3.2 Optimization techniques. Various techniques for robustness optimization differ in the way they guide the input sampling (search) process and can largely be divided into "active" techniques and "passive" techniques [23].

In "active" optimization techniques, dominated by Bayesian optimization [24], one tries to learn a (probabilistic) model of the input-to-robustness function, to select the next-best input to test. These techniques often rely on a Gaussian Process (GP) in order to learn this model, and require making numerous assumptions on the underlying process distributions. While some common assumptions

seem to work well in many cases, these are often not grounded in actual knowledge, but are chosen to ease the computation. Bad choices might hinder convergence—a property that is, regardless, only guaranteed for mappings. As shown, e.g., in [25], the optimization performance can drop significantly when dealing with complex or composite specifications, which might compromise the smoothness assumptions. These cases thus require usage of dedicatedly-crafted optimizers that use expert knowledge to effectively decompose the specification (e.g., [26, 27]). While it is potentially possible to bias active techniques with domain knowledge, this knowledge is typically expected as prior sampling distributions (as in [28]), which is often not trivial to achieve and unavailable.

"Passive" optimization techniques, often referred to as "direct search" or "random search" methods, rely on various non-learning techniques to ensure the sampling procedure of the inputs well-covers the space. These include, for example, straightforward sampling approaches like Line-Search [29] and Simulated Annealing [30, 31], or more advanced ones, like the Cross Entropy Method [32, 33]—a type of importance sampling in which one iteratively estimates a sampling distribution based on the output values (but does not estimate a model of the system); this approach, like Bayesian optimization, also requires setting assumptions on the underlying distributions. Notably, all of the aforementioned optimization techniques assume the input space is defined in a continuous box, and do not natively support discrete input variables. Although seemingly less popular today, evolutionary (genetic) algorithms were also examined as a direct search technique [34, 35]. In such a serial approach, one starts from a "population" of $n \geq 1$ samples (rather than a single one, in the previous approaches); then, in each step, a new generation of n samples is created, through crossovers (mergers) and mutation among the previous population. These evolutionary methods precede the notion of robustness, and can be used to optimize a general objective function (provided by the user as domain knowledge); they may also generally support discrete variables.

In addition to the system-independent approaches mentioned above, many works provide customized solutions for specific systems. For example, [36] focused on testing of autonomous driving systems. That work discusses the scenario of falsifying the trajectory of an autonomous car, though in a somewhat different setting than the one we use in our running example (they search for a sequence of runtime interventions from other agents, while we search for an input scene). The contribution of that system-and-scenario-specific work, and those like it, is explaining how that falsification problem can be formulated to best fit an existing falsifier; in that case—a genetic-algorithm-based falsifier. It is worth mentioning that several recent works [37, 38], focus specifically on robustness optimization for systems with NN components. These works suggest various ways to use feedback from monitoring of the NN inference process/outputs, in order to guide future sampling and the overall optimization process. We do not assume access to such expert knowledge.

1.3.3 Planning-based falsification. Beyond the optimization approach covered, another prominent approach for solving the falsification problem is by formulating it as a path/motion planning, and employing sampling-based motion planning algorithms [39, 40] to solve it. Such algorithms are celebrated in robotics research for their ability to effectively explore high-dimensional state spaces (e.g., configuration spaces of high-DoF manipulators). In contrast to the aforementioned optimization-based techniques, in which one samples complete inputs for the system and use them to generate full system-trajectories to test, these approaches (e.g., [41–47]) build system trajectories incrementally, by interleaving sampling of local disturbances as partial system inputs (e.g., discrete control inputs, system noise, or actions of other agents) and local, time-bounded system simulations. They treat the injection of disturbances as a mean to control the system trajectory, and use planning techniques to effectively grow a tree of diverging trajectories, in hope to find a falsifying one. The specification validity is checked on each trajectory prefix, after each extension, until a trajectory with violation is detected. These approaches are thus more relevant

to controllable hybrid-systems or reactive systems with dynamic disturbances, and not, e.g., to verify robustness to varying environments/initial conditions. They are also, by their nature of operation, limited in the type of specification they can falsify: they are designed to handle "safety specifications," in which the system is safe by default until at some point (after incurring a certain disturbance) it becomes unsafe; they are not designed to handle "liveness specifications" [48], in which a partial system trajectory does not satisfy the task, until at some point it does. The latter case is especially relevant for controllers of robotic systems performing abstract tasks, defined, e.g., as symbolic reach-avoid constraints, PDDL goals [49], or finite-LTL formulae [50].

Other approaches that, in a similar fashion, also incrementally build a diverging system trajectory tree were also examined, including Monte Carlo Tree Search (MCTS) [51], a reward-based tree growth technique, and (deep) RL [52, 53], which uses learning to generate a policy to guide the tree growth.

1.3.4 Parameterization of the input space. As mentioned in the beginning of this review, to automatically generate test inputs, falsification techniques must consider some sort of finite input parametrization. It is ubiquitous to assume the values of these parameters are limited to continuous ranges ("boxes") [24, 29, 32, 34]. In optimization-based approaches, the number of parameters must also be predefined and set. Further, most existing techniques and tools assume the test input to be a signal that can be presented as a parameterized curve [24, 32, 34, 54] or a discrete sequence of disturbances [18], and that these parameters are externally controlled, and thus can be sampled arbitrarily [55, 56]. While these assumptions might be reasonable for testing controllable systems, where the control-signal is often of a numeric, low-dimensional, free-to-choose vector, they are less so for autonomous systems, where a realistic sensor-signal is often high-dimensional and cannot be sampled validly without simulating the system (due to its continuous dependence on the state). As a result, making such assumptions imposes severe limitation on the systems we can analyze and techniques we can use. Indeed, existing works on falsification of autonomous systems typically consider very simple scenarios, such as an adaptive cruise control systems [37], a multi-aircraft conflict-resolution system [43], an Abstract Fuel Control (AFC) system [57], an insulin infusion system [32], or a goal-reaching navigation system [25]. In these systems the inputs are simple: initial conditions represent the initial system state, and the input signal describes *observations of an external quantity*, given as a continuous, low-dimensional signals, which can fluctuate freely, e.g., in the mentioned examples, the position of the leading car, the direction of wind, the engine speed, the blood glucose level, or added motion noise.

A few recent examples [1, 52, 58, 59] mitigated some of these issues by modeling the test input as a set of rich initial conditions, or a description of "an environment," used to initiate a simulation, which could then generate the sensor signal (instead of assuming it can be sampled directly). We support this concept and find it to be more appropriate for general autonomous systems.

1.4 Our approach: falsification as "meta-planning"

1.4.1 Standardized environment-based testing model. As mentioned, instead of trying to reason over sensor-observation signals as the test inputs, we believe a more general and more appropriate model should consider *environments* as inputs. Thus, as the first part of our contribution, we formulate and standardize a testing model that considers an environment as input (as illustrated in Fig. 2). An environment is a description of all variables *external* to the system that can affect the system throughout its run (or the specification). The formalism we suggest extends the concept of rich initial conditions and is specifically intended to support systems operating in and observing rich, modular, open-world environments. The environment, together with the state of the system in it, would determine the observation stream, which would be generated incrementally through a simulation.

The environment-based model may be viewed as an alternative input parameterization to the signal-based standard. Accordingly, it may be utilized with the previously-covered falsification techniques, enabling their usage for autonomous systems—something that is not natively supported when considering standard models; we demonstrate this aspect in our experimental comparison.

1.4.2 Falsification as meta-planning. As the main contribution of this paper, and utilizing the suggested environment-based input parameterization, we propose a new direct-search falsification approach for autonomous systems under general formal specification. In our approach, we reformulate the falsification problem, the search for inputs that would fail the system, as the problem of planning a trajectory for an (under-actuated) meta-system, which wraps and encapsulates the examined system; we call this approach: meta-planning. *Each meta-state encapsulates both a system input (environment) and its corresponding output (a simulated system trajectory in the environment).* A meta-control, which can be applied on meta-states, invokes a local change to the input and a corresponding update to the system trajectory. The goal meta-region contains all the meta-states in which the system trajectory indicates failure of the specification. This formulation allows us to apply standard and efficient planning techniques to search the meta-state space and discover a meta-trajectory that leads to a goal meta-state—a solution to the falsification problem. Note that the objective of meta-planning (unlike the traditional usage of planning) is simply to find a goal meta-state, corresponding to a falsifying example; the meta-trajectory, which represents the sequence of environment mutations from an initial guess into a falsifying one, is less important. We emphasize that this contribution is not a new search algorithm, but a *reformulation of the falsification problem,* which enables its efficient solution using existing tools. In general, meta-planning does not *require* any domain knowledge; with that said, it can incorporate and benefit from a variety of domain knowledge sources, if they are available. This general falsification technique is, in fact, applicable to falsification of any black-box system, beyond merely autonomous systems, as it only requires input-output knowledge. Yet, as a unique feature and contribution, when considering an autonomous system, we show that meta-planning can be performed using only incremental simulation, to minimize redundant calculations for each test case, and further improve its efficiency.

1.4.3 Summary of contributions. To summarize, this paper contributes (i) a standardized testing model for autonomous systems operating in a rich, open environment (Sec. 2), in support of (ii) a novel formulation of the falsification problem as a meta-planning problem, (Sec. 3). We also provide (iii) a practical explanation and guidelines on how to solve the meta-planning problem using sampling-based motion planning algorithms (namely, RRT), and how to potentially incorporate domain knowledge to accelerate the search (Sec. 4). Finally, we present (iv) an experimental study on using meta-planning for falsification of an obstacle-avoiding autonomous car with a NN controller, including a comparison against alternative falsification approaches, in which meta-planning shows superior performance (Sec. 5). Pseudocode summary of our algorithms is provided in Appendix A, and extended theoretical discussion and comparison is provided in Appendix B.

2 Standardizing the testing model: autonomous systems in uncertain environments

We care to examine a simulatable, autonomous, dynamical system, referred to as the "system," acting in an *environment*, which encodes the external factors affecting the system's operation.

2.1 Environment formalism

Formally, we define the *environment-type* as the set of properties or variables one should specify a value to, in order to describe the context in which the system operates. An object of that type, which determines a specific valuation to these variables, is referred to as an *environment-state*. An environment-state can be observed by the system, and potentially changed by it.

Variables can be classified as either environment *parameters* or *elements*—a separation we will later exploit for computational gains. Simply put, parameters describe (i) essential information, which (ii) affects the system's observation and/or dynamics globally, regardless of the system state; elements describe (i) optional, additive features used to enrich the environment, which (ii) may be observed locally, from only a subset of system states (making the environment partially-observable by the system). The environment-type defines for both parameters and elements their type, e.g., a scalar, a vector, a function, or even a time-dependent model. By convention, elements should be organized into *collections*, grouped together by the features they describe; such collections may be ordered (i.e., a vector of elements) or unordered (i.e., a set of elements), and may also indicate the minimal/maximal number of elements that may be provided, or other restrictions, e.g., on the mutual-exclusiveness of the elements. This formalism we suggest is not limited to continuous variables in box-ranges, but can include discrete and symbolic variables, and even be used to encode time-varying perturbations. As we use collections to specify the environment elements, the number of variables in each environment-state is not predetermined—allowing us to model systems operating in rich, open environments. We provide a concrete example at the end of this section.

2.2 System model

While we support both continuous and discrete systems, for generality, we assume the system operates under continuous dynamics, modeled as: $\dot{\xi} = f(\xi, u, env), \quad u = g(z), \quad z = h(\xi, env)$, where $\xi \in \Xi$ is the system state, $env \in Env$ is the environment-state of environment-type Env, $u \in U$ is a control action, $z \in Z$ is a sensor observation, $f : \Xi \times U \times Env \to \Xi$ is the vector field, $g : Z \to U$ is a "black-box" NN-controller, and $h : \Xi \times Env \to Z$ is the observation (sensor) model. Note that while the environment-state can encode time-variant variables, its *definition* is assumed to be static throughout the system operation. We use ξ_t to mark the system state at time $t \in [0, T]$, and $\bar{\xi}_{0:T}$ to mark the concatenation of states from time 0 to time T, a time-parameterized, continuous trajectory. If the length of the trajectory is not of interest, we may drop the subscript notation.

Note that, **for conciseness of discussion, we will consider the initial system state ξ_0 to be given and set**, and the system to contain neither actuation nor sensing noise, making the environment-state the only controllable simulation input, and the environment-state variability the only source of uncertainty. We will also assume the control input is updated at some constant frequency. These assumptions are not essential to our approach. Moreover, from now on, wherever the context is clear, we will simply use the term "environment" to refer to an environment-state.

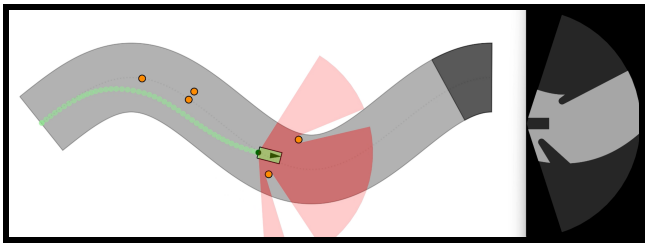


Fig. 1. Our running example: an autonomous car acting in an "obstructed track" environment, whose *parameters* define the track shape, while its *elements* are the obstacles, which are organized in the "obstacle collection." The car trajectory is in green. At each state, the car observes the track using a lidar sensor; the observation image of the area highlighted in red is shown on the right. The car NN-controller is trained to take in the stream of observation images and steer the car to the end of the track without collision.

2.3 Task specification

We assume the NN-controller was trained, e.g., using RL, to guide the system towards completing a *task*—defined as a constraint over the system trajectory—while being robust to perturbation; we make no assumptions on the NN or the way it was trained. Unlike standard approaches, we allow the task constraint to *also involve the environment*. The task may be defined in various ways, e.g., with a temporal logic formula, or using other types of logic, and may express both safety and liveness specifications. To evaluate the system's success in a task, we assume availability of a "status" predicate: $\texttt{status}(\bar{\xi}, env) \in \{0, 1\}$, which encapsulates evaluation of the task constraint, and abstracts away the specification's specifics. Evaluation of this predicate can be done with an appropriate model checker. For a trajectory $\bar{\xi}$, resulting from a system run in an environment *env* over a period of time, `status` returns 1, if the trajectory satisfies the task, or 0, if it does not.

2.4 Controller falsification: problem definition

As means of verifying the behavior of the system and the controller's robustness, our objective is to try to falsify the specification. Meaning, we wish to find and return a witness of failure, i.e., an example of an environment *env*, and a system trajectory $\bar{\xi}$, which results from running the system in *env*, for which the task specification is violated—if such a witness exists. While we assume we can easily observe whether the system trajectory conveys a success or failure of the task, if the controller is well-trained, it might not be trivial to *find* a scenario in which the system fails. Thus, we specifically care to falsify the controller using minimal simulation effort (length), which we may measure using the number of control loops simulated. Further, we would like to rely on a system-agnostic falsification technique, which minimizes the reliance of expert knowledge.

2.5 A running example: an autonomous car

While the ideas presented in the following sections are relevant to a general system, to ground the discussion, we will consider a running example of "an autonomous car on an obstructed track," as depicted in Fig. 1.

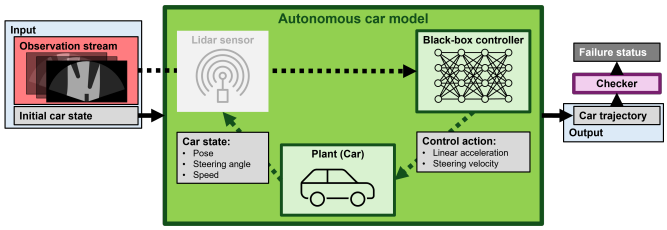
For an environment of type "obstructed track," the variables are: track curve, track range, track width, and a collection of obstacles. The first three variables are environment parameters of the types: function $f\colon \mathbb{R} \to \mathbb{R}$, vector $[x_{\text{start}}, x_{\text{end}}] \in \mathbb{R}^2$, and non-negative scalar $\in \mathbb{R}^+$, respectively; the fourth is a set of "circular obstacle" elements, each of which of the type $[x, y, r] \in \mathbb{R}^2 \times \mathbb{R}^+$. Each environment definition induces several regions in the xy plane (over which the track is laid), including the track end zone, the obstructed area (the area covered by the obstacles), and the track shoulders (the area beyond the edge of the track).

The state $\xi \doteq [x, y, \phi, \alpha, v]^T \in \Xi$ defines the car's origin position and heading (which together make the car pose, from which we can derive that of the car's overall geometry), its steering angle, and speed. The observation model h returns a lidar scan of the track from the car's pose (an image). The controller g sets the acceleration and steering velocity, given the stream of lidar scans (and the concurrent steering angles), and f complies to a bicycle model. The controller was trained to steer the car to the end of the track without collision, regardless of the track curve and obstacle placement. This task can be defined with the finite LTL (LTLf) [50] formula

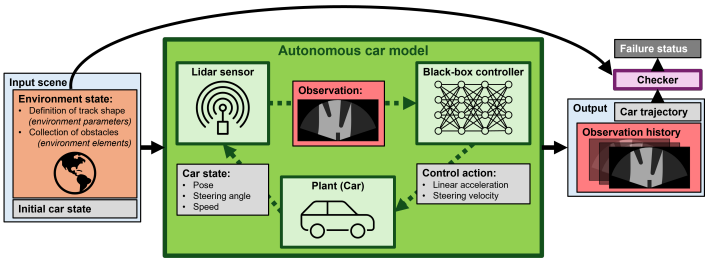
$$\Diamond(\xi.shape \subseteq env.end\_zone) \wedge \Box\neg(\xi.shape \cap env.obstructed\_area) \wedge \Box\neg(\xi.shape \cap env.shoulders), \quad (1)$$

which can be encapsulated in a `status` predicate, which returns 1, if the car's trajectory satisfies the formula, and 0, otherwise.

Our goal in this scenario is to efficiently find an obstructed track environment in which running the autonomous car results in collision.



(a) A standard-yet-inappropriate testing model for the autonomous system, attempting to model the sensor observation stream as input.



(b) Our suggested amended testing model, considering a scene as input, and the sensor observation stream as the simulation output.

Fig. 2. Testing models used for falsification of autonomous car system.

2.6 The testing model

As previously covered, typically, the observation trajectory is assumed to be externally-controllable. Hence, the standard testing model used for falsification is based on parameterization of the sensor signal as a test input. Yet, here, we do not make this assumption, but consider the sensor observation to be a function of the environment-state and the system state. This means that to generate a temporally-consistent sensor signal, it must be generated incrementally by running a simulation of the autonomous system, at the same time the system trajectory is generated. This property is amplified when considering high-fidelity observations, like a camera stream, which can only be generated by the simulator. Accordingly, the standard testing model, in which the sensor signal is considered an input, is simply inappropriate for testing our scenario, as depicted in Fig. 2a.

To mitigate this gap, we suggest a slightly-but-crucially amended model, based on our environment formalism, as presented in Fig. 2b. In this model, the simulation and test input is defined by an environment-state *env* and an initial system state ξ_0, which together comprise a *scene*[1]. The simulation output is the system trajectory $\overline{\xi}_{0:T}$, alongside the sensor observation history $\overline{z}_{0:T}$, which are then fed to a model checker, to generate the test result. As we shall see, including explicitly the observation history as a simulation output will allow us to exploit the concept of incremental simulation when analyzing the system. This model also supports more general task specifications, which depend also on the environment (beyond the system trajectory), as suggested before.

3 Main contribution: reformulating the falsification problem as "meta-planning"

Our goal in falsification is to search the input space for one for which the corresponding output indicates specification failure. In the case of an autonomous system, as we previously modeled, this means searching the scene space, i.e., all possible initial states × all possible environment-states of the relevant environment-type, for a scene in which the system would not satisfy its task. Instead of serially sampling inputs and performing independent tests, we suggest to start from an initial test,

[1]This terminology is consistent with standard falsification tools like Scenic [60].

and search by performing sequences of gradual, local changes to it, until one of these sequences leads us to a test that satisfies the condition. As we introduce ahead, this approach to falsification can be formulated as planning of a path to a "meta-system," which wraps the examined system.

3.1 Meta-planning in the meta-state space

We mentioned that, to solve the falsification problem, we should search the scene (input) space. Yet, we note that the goal of the search is defined over the system trajectory (output). By such, for each input that we reach in our search, we must also simulate the corresponding output, in order to validate its status or measure the search progress. This means, in fact, that when searching the input space, we implicitly search the composite space of of inputs × outputs. We suggest here to reason about and search this composite space *explicitly*, as this could allow us to better guide our search—by analyzing the diversity of both the sampled environments (like optimization-based approaches) *and* the system reactions (like planning-based approaches). In our case, this means explicitly reasoning about and searching the composite space $\overline{S} \subset$ environments × system-trajectories × observation-histories, containing all valid *simulated scenes*, where a *simulated scene* $\overline{s} \doteq (env, \overline{\xi}_{0:T}, \overline{z}_{0:T})$ is said to be *valid*, if the system trajectory $\overline{\xi}_{0:T}$ and observation history $\overline{z}_{0:T}$ are consistent with the system's observation, controller, and transition models, and environment *env*. In other words, the simulated scene is valid if running the system in this scene produces (or can produce, for non-deterministic scenarios) the given system trajectory and observation history. The space $\overline{S}$ shall serve as the "state space" in our planning-based formulation. Crucially, since we do not plan a trajectory for the system, but plan in the space of system trajectories, we refer to this problem as *meta-planning*, to simulated scenes as *meta-states*, and to $\overline{S}$ as the *meta-state space*. The status predicate can be restated accordingly as a predicate on meta-states: $\texttt{status}(\overline{s}) \in [0, 1]$.

3.2 Meta-control

To transition between meta-states, we need to define meta-control actions. Each meta-control action corresponds to a scene mutation, followed by a simulation of a system run in the mutated scene, in order to update the system trajectory and determine the new meta-state. Note that, since we consider here the initial state to be set, we may more simply consider only mutating the environment, as we shall do from now on. In general, mutation may also affect the initial state.

3.2.1 Environment mutation. An environment mutation is a transition from one environment-state to another in the environment-space induced by an environment type (e.g., all obstructed tracks). **For the interest of this paper, we restrict the mutation only to changes to the environment-element collections,** excluding the environment parameters (and the initial state). In our running example, a legal mutation would convey a change in the set of obstacles placed on the track, without changing the track itself.

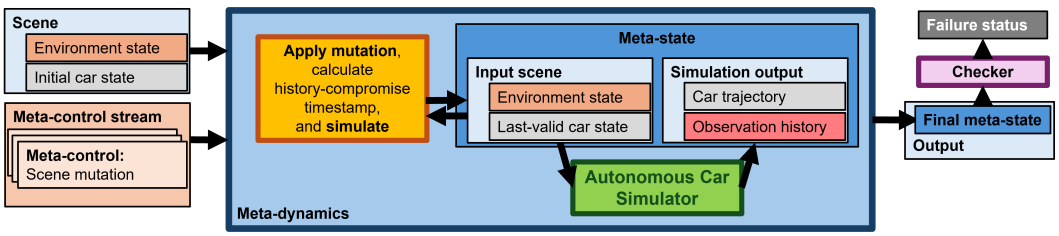


Fig. 3. The meta-system model.

To specify a mutation, we use a 3-tuple of the form $mut \doteq (coll, op, elements)$ where $coll$ is the name of the element collection, op is the operation to be performed on this collection, and $elements$ are inputs to this operation. We consider two basic operations: $\oplus$ and $\ominus$, which indicate *addition* and *subtraction* of elements to/from a collection, respectively. The result of applying the mutation $(coll, \oplus, elements)$ (respectively, $(coll, \ominus, elements)$) on an environment env is a new environment, marked $env \oplus_{coll} elements$ ($env \ominus_{coll} elements$), in which the value of the collection $coll$ is $env.coll \cup elements$ ($env.coll \setminus elements$), and the value of all other variables is the same as in env. More generally, we may consider the *replacement* operator $\oplus$, which can remove elements from a collection and add others in their place. This operation, marked $\oplus$, can be defined by combining of $\ominus$ and $\oplus$: $env \oplus_{coll} (old\_elements, new\_elements) \doteq env \ominus_{coll} old\_elements \oplus_{coll} new\_elements$. For example $env \oplus_{obstacles} (\{(x_1, y_1, r_1)\}, \{(x_2, y_2, r_2)\})$ indicates removal of the obstacle (x_1, y_1, r_1) from the obstacle collection of env, and its replacement with (x_2, y_2, r_2).

3.2.2 Meta-dynamics. The space of legal mutations represent the meta-control space. Meta-controls can be applied on meta-states, which are updated according to the "meta-dynamics:" $\mathfrak{F}(\overline{s}, mut) \doteq \overline{s}'$, where $\overline{s} \doteq (env, \overline{\xi}, \overline{z})$ is the input simulated scene; $mut \doteq (coll, op, inputs)$ is the selected mutation; and $\overline{s}' \doteq (env', \overline{\xi}', \overline{z}')$ is the output simulated scene, where env' is the mutated environment, and $\overline{\xi}'$ and z' are the new system trajectory and the observation history, respectively, derived from a new system simulation in env', and complying to the agent models, i.e., $\dot{\xi}' = f(\xi', g(z'_t))$, $z'_t = h(\xi', env')$.

3.3 Falsification as meta-planning for a meta-system

Together, the meta-state space and meta-control space we defined comprise a meta transition-system, which wraps and encapsulates the examined system. We note that, as the system trajectory is returned from a simulator, we do not have direct control over all meta-state variables—we can only actively manipulate the environment and then observe the effect on the system trajectory. In a systems perspective, this means our meta-system is "under-actuated." Further, since we consider a black-box controller, this means we do not have an analytical model for the meta-dynamics. We also note that, by this definition, this meta-system is a discrete and Markovian transition system, even when considering a continuous system (as a mutation and re-simulation is a discrete transition, and every posterior meta-state only depends on the prior meta-state and the applied mutation).

We are interested in returning a meta-state that satisfies the condition $\texttt{status}(\overline{s}) = 0$. This condition defines our region of interest within the meta-state space, which contains all the witnesses of task failure. While this condition is easy to test for a given meta-state, it is not trivial to generate a meta-state from this region. To solve this problem, we suggest starting from some initial meta-state for the meta-system, and then trying to lead it into the region of interest, representing a goal region. Accordingly, we can present the falsification problem as the motion planning problem of finding a goal-reaching trajectory for the meta-system, whose states are simulated scenes.

Problem definition. Starting from an initial meta-state $\overline{s}_{init}$, find a sequence of meta-controls whose application leads the meta-system into a goal meta-state $\overline{s}_{goal}$, at which $\texttt{status}(\overline{s}_{goal}) = 0$.

As we will see next, this formulation (visualized in Fig. 3) enables us to use efficient motion-planning techniques to search for a solution—potentially by evaluating fewer inputs than we would have by using independent sampling. We clarify, however, that, unlike typical motion planning, in which we are interested in the *path to the goal region* (and often, an optimal one), the solution to our problem is not the path, but the *goal state itself*, which expresses the falsifying example. We are hence only interested in finding *a* solution, as fast as possible. The path from the initial meta-state to the goal meta-state simply expresses the solution (environment mutation) process, until converging to a falsifying witness. Further, this motion planning problem does not contain

explicit "obstacles." While we could maybe consider the invalid meta-states (where the system trajectory is not consistent with the environment) as "obstacle regions" to avoid, still, by definition, starting from a valid state and applying meta-controls, these cannot be reached.

We should also emphasize that this high-level falsification approach is applicable to general black-box systems, with general inputs and outputs, and not just for autonomous systems; though, to keep the discussion grounded, we will continue by considering the case of autonomous systems, where the inputs and outputs are as described. Adaptation to other systems is straightforward.

3.4 Exploiting incremental simulation for improved per-test efficiency

As we recall, we would like to minimize the amount of simulation effort needed to find a falsifying meta-state. Our reformulation of the problem as meta-planning should help us, as we explained, to reduce this simulation effort, by reducing the number of tests (simulations). Yet, when testing autonomous systems in rich environments, the incrementality of the test-case generation, which characterizes meta-planning, can *also* allow us to reuse simulation effort across tests; by such, we can also improve the per-test efficiency and the overall falsification efficiency even further.

Environments, according to our formulation, are enriched with collections of environment elements (e.g., obstacles). Since elements, unlike parameters, are only locally-observable by the system (e.g., only from "nearby" states), mutating the elements in a simulated scene may, in general, only partially compromise the validity of the simulation output—only starting from the timestamp in which this mutation was observed. This, in turn, means that system trajectories in consecutive meta-states (i.e., meta-states connected via application of a meta-control) share an *overlapping prefix*. This property, illustrated in Fig. 4, can be exploited to avoid redundant calculations when applying a meta-control and reduce the simulation effort needed to test each new input during our search. To achieve this, each meta-control application should only invoke partial simulation, to update only the *suffix* of the trajectory, instead of simulating the system from scratch. This can be done practically by first rolling back the original simulation, performing the environment mutation, and then continuing the simulation. We can now see that maintaining the observation history as part of the meta-state is essential to understanding the timestamp from which the system trajectory is compromised. Accordingly, we amend the previously-defined meta-dynamics model $\mathfrak{F}$ to involve only an *incremental update* of the system and observation trajectories: $\mathfrak{F}(\overline{s}, m) \doteq \overline{s}' = (env', [\overline{\xi}_{0:T}, \overline{\xi}'], [\overline{z}_{0:T}, \overline{z}'])$, where T is the last timestamp where $h(\xi_t, env) = h(\xi_t, env')$, and $\overline{\xi}'$ is a trajectory that starts at ξ_T and complies to the system models in the mutated environment env' (i.e., the result of starting the simulation from a scene (ξ_T, env')).

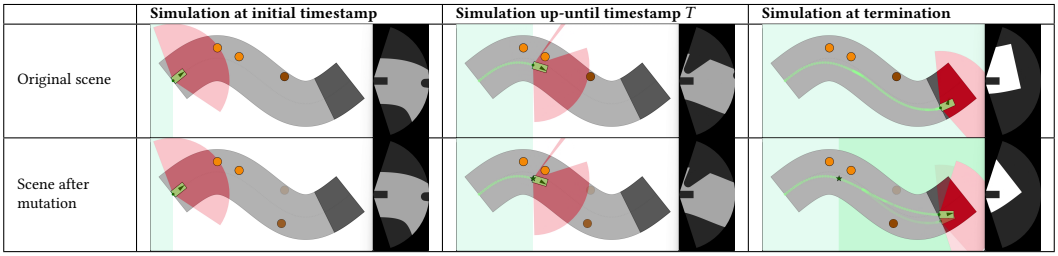


Fig. 4. Comparing the simulation of a scene (environment and initial state) and of the scene after locally mutating it by moving the brown obstacle. The simulation progress is indicated using the green background. The environment mutation only affects the system trajectory starting from timestamp T, at which the mutation was observed, and the original observation history was compromised. Thus, to evaluate the mutated scene (calculate the new meta-state), it is enough to run a partial simulation, from the car state at timestamp T.

We can practically calculate T in a domain-independent way, without making any assumptions on the sensor/controller, by using minimal to no simulation effort. To do so, we should sequentially go over the states in the original trajectory $\overline{\xi}$, use the simulator to generate a new sensor observation z'_i from each state $\overline{\xi}_i$ in the mutated environment env', and compare this z'_i to the original observation z_i taken in the original environment env; if they do not match, it means the history is compromised at this timestamp, otherwise, we should continue checking the rest of the trajectory. If the system is uncertain, we can, more generally, check if it is possible to observe z_i in env'. Note that generating the observations in the trajectory prefix would have been *an inherent part* of the simulation, had we performed it. This means that running this "overlap-identification" procedure is always more efficient than (and preferred to) running the prefix simulation, as it spares all other simulation components, e.g., of the system dynamics or the (NN) controller. Furthermore, this procedure can oftentimes be performed without simulating anything at all, but through simple analysis, given appropriate knowledge on the sensor model. E.g., in our example, we can validate each observation z_i by first calculating (or conservatively estimating) the "observed area," that is, the observed portion $\subseteq \mathbb{R}^2$ of the track, based on the sensor properties (range and angle), the state $\overline{\xi}_i$ from which the observation was taken, and the environment env; then, simply check if any of the added/removed obstacles overlap with that area, compromising the observation. The generic and domain-specific versions of this overlap-identification procedure are both summarized in Alg. 3, in Appendix A.

4 Solving the problem: sampling-based forward search in the meta-state space

As our innovation is in the reformulation of the falsification as a (meta-)planning problem, **our flexible approach is not tied to a specific planner**. We can potentially use various off-the-shelf planners to solve the meta-planning problem. Nevertheless, since the state space and control space of our meta-system are infinite, to solve the motion planning problem, we suggest to resort to Sampling-Based Motion Planning (SBMP) approaches [39, 40]. Further, as we recall, we do not have an explicit model for the meta-dynamics of our meta-system and must rely on a simulator to apply meta-controls. Thus, we do not have access to a "steering function" or a "local planner" often used by such planners to solve the Two-Point Boundary Value Problem (TPBVP) (i.e., finding local meta-controls connecting two meta-states) [61]. We hence cannot use techniques that rely on backward planning, nor on road-map-based planners. Besides that, road-map-based approaches rely on drawing numerous samples from the (meta-)state space, which, in our case, is equivalent essentially to the "naive" falsification approach, in which we sample and test inputs independently; while backward planning relies on having access to a goal-state, which we clearly do not have. Thus, overall, to solve the meta-planning problem using SBMP, we may **focus only on planners that build a forward-search tree in the (meta-)state space**, as illustrated in Fig. 5.

Sampling-based forward-search planners [62] rely on two sub-procedures, which we call alternately for building the search tree: (i) tree-node selection, and (ii) tree-node expansion. Forward-search planners differ in the way they implement these sub-procedures. The rest of this section will provide guidance on how we can implement these sub-procedures in our falsification setting, and how we can effectively incorporate basic domain-knowledge into them, to accelerate the search. Procedures for node selection and expansion are summarized, respectively, as Alg. 2 and Alg. 3 in Appendix A; there, the various options detailed ahead may be enabled by the user by setting respective flags (highlighted in red). We summarize the overall algorithm for meta-planning-based falsification using sampling-based forward search as Alg. 1, also in Appendix A.

4.1 Node selection: biasing the search towards goal

Unguided strategies for selection of nodes (meta-states, to be expanded), such as Breadth/Depth-First-Search, or random selection, might take a long time to converge to a solution. To guide the

search towards the goal, and hopefully accelerate it, we can greedily prioritize the selection of nodes for expansion based on their distance-to-goal. In our context, in which the goal of the search is a witness of failure, a distance-to-goal of a meta-state should indicate how "close" (we think) the corresponding system was to failing the task, based on its trajectory; we refer to this as the *distance-to-failure* heuristic. This heuristic can be conveniently encoded by overloading the `status` predicate: instead of a binary output, we can allow it to return a value in the range $[0, 1]$ conveying the (normalized) heuristic value. It is possible but surely not required to use the robustness function as the heuristic. In general, this heuristic does not need to be smooth or in any restricted format.

In our running example, an intuitive distance-to-failure heuristic can be derived by examining all the states in the car trajectory, finding and returning the car's minimal *distance-to-collision*, i.e., the minimal Euclidean distance of the car shape from an obstacle or the track shoulders along its trajectory. It is possible to choose a different heuristic.

Finally, note that some planners (e.g., RRT [63], as we discuss next) typically rely on sampling of a new goal state in each "goal-biased" algorithm iteration, and measuring a state-to-state distance from tree nodes to it, in order to estimate their distance-to-goal (and rank them for selection). Such direct goal-state-sampling, as we recall, is not possible in our case; thus, even if planning with such a planner, we would still rank the nodes based on a distance-to-failure heuristic, which conveys the estimated distance to the goal *region*.

4.2 Node selection: adding exploration

While beneficial, relying exclusively on a greedy node selection strategy might cause us to get stuck in a dead-end of a (distance-to-failure) local minimum. So, to ensure complete and efficient solution, it is important to allow for exploration in node selection. As a parameter of the planning algorithm, we can define a ratio between iterations of greedy node selection, and of exploratory selection (according to some strategy), known as the "goal bias." The most basic exploration strategy is to use random sampling of tree nodes during exploration iterations. This technique requires no domain-knowledge, though it might not promote a very effective exploration of the search space. Instead, modern sampling-based motion planners rely on domain-specific information, e.g., a state-to-state distance function and a state-sampling procedure by RRT [63] and EST [64], or a state projection function for density estimation by KPIECE [65], to guide the search and ensure active exploration of the search space. If available, providing such domain-knowledge would allow us to use more advanced, standard planning techniques for an efficient solution.

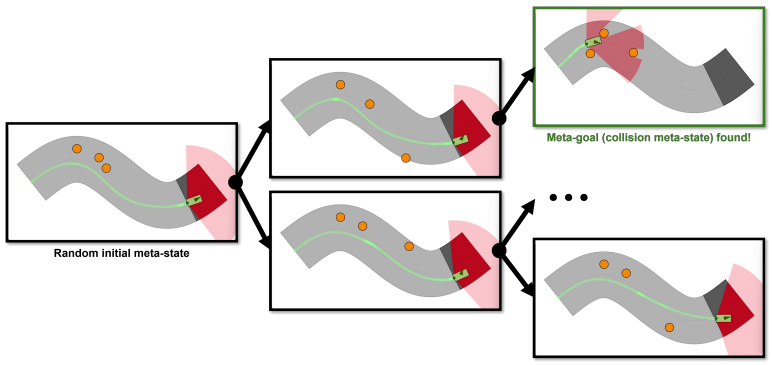


Fig. 5. Solving the meta-planning problem: a forward search-tree, where meta-states (nodes) represent simulated scenes, and meta-controls (edges) represent scene mutation followed by incremental re-simulation.

4.2.1 Measuring distance between meta-states. Let us discuss how we can define a meta-state distance function to enable usage of planners like RRT or EST. In the context of EST, this function shall be used to estimate the density of expanded nodes, in order to prioritize nodes in less-explored regions. In the context of RRT, this function shall be used to measure the distance between nodes in the search tree to a randomly-sampled meta-state; we would then select for expansion the tree-node closest to the sampled meta-state, as a technique for growing the tree toward less-explored regions.

Regardless of our specific problem domain, we can use a composite distance function of the following form between two meta-states $\bar{s}^1, \bar{s}^2$:

$$\texttt{meta\_state\_distance}(\bar{s}^1, \bar{s}^2) \doteq w \cdot \texttt{env\_distance}(env^1, env^2) + (1 - w) \cdot \texttt{traj\_distance}(\bar{\xi}^1, \bar{\xi}^2), \tag{2}$$

where $w \in [0, 1]$, the first component accounts for the distance between the respective environments, and the second accounts for distance between the respective system trajectories.

We recall that our formulation assumes that meta-controls can only modify each environment's elements (e.g., obstacles), which are organized in collections (e.g., sets of obstacles). Thus, we can define the following environment-distance function, which, for a pair env^1, env^2 of environments of the same type *Env*, only accounts for the difference between their respective element collections:

$$\texttt{env\_distance}(env^1, env^2) \doteq \sum_{coll \in \text{ element collections of } Env} \delta_{coll}(env^1.coll, env^2.coll), \tag{3}$$

where for each element collection *coll* defined in *Env*, δ_{coll} is a distance function defined between the corresponding collections $env^1.coll$ and $env^2.coll$.

Although we may use a simple, domain-agnostic set-difference function, the function δ_{coll} would benefit from a domain-specific implementation. In our running example, this function should represent the distance between sets of obstacles on the track, corresponding to env^1 and env^2. As an example, we chose a geometric function aimed at estimating the overlap between the two sets in the 2D plane. The function is calculated as follows: first, we choose a random point inside a random obstacle from the first obstacle set and measure its Euclidean distance to its nearest obstacle from the second set; we repeat this process multiple times and average the distances, to overall estimate the distance of the first set from the second. Since this distance is not symmetric, we should use a similar process to calculate the distance of the second set from the first and, finally, return the average between the two. It is possible to choose a different distance function.

To complete the meta-state distance function, we require a distance function $\Delta(\bar{\xi}^1, \bar{\xi}^2)$ between system trajectories. In the context of our running example, since the trajectories can, generally, be of different length, we cannot simply rely on measuring distance between corresponding states. Yet, since the system motion is continuous and the states are geometric, we may use, for example, a function based on distance-between-curves. We chose to use the 2D area bounded between the curves, calculated by estimating the integral of the difference between the curves (where both trajectories are normalized to the span the same length of time):

$$\texttt{traj\_distance}(\bar{\xi}^1, \bar{\xi}^2) \doteq \int_{t_{\text{init}}}^{t_{\text{final}}} \left[\max(\bar{\xi}^1(t), \bar{\xi}^2(t)) - \min(\bar{\xi}^1(t), \bar{\xi}^2(t)) \right] dt. \tag{4}$$

4.2.2 Sampling of meta-states for planning with RRT. Generally, sampling a meta-state is done by sampling a scene and running a simulation of the system. In our running example, where the initial system state and environment parameters are set, sampling a scene simply means sampling locations for the obstacles on the track. We note again that typical RRT explicitly distinguishes between sampling of goal (meta-)states during goal-biased algorithm iterations, and of non-goal (meta-)states during exploratory iterations. Yet, in the case of our meta-system, we cannot make

this distinguishment, and can only sample random meta-states (otherwise, we could have solved the falsification problem). As already mentioned, for that reason, in each goal-biased algorithm iteration, we should rely on the distance-to-failure heuristic for node selection, instead of measuring the distance to a sampled goal meta-state. At each exploratory algorithm iteration, we should simply sample a random meta-state and then check whether it is a goal meta-state or not: if it is not (the likely case), then we would extend the tree towards this meta-state, as normal; if we discover it is goal meta-state, then we actually *found a solution* to the falsification problem, and we can quit the search. This altogether means, somewhat unconventionally, that RRT-based meta-planning (i) uses two different distance functions: one to grow towards the goal region, and second to grow towards random samples; and (ii) adds another possible termination option, during node sampling.

4.3 Node selection: simplified

4.3.1 Simplified distance between meta-states. We recognize that measuring system-trajectory distance (in order to calculate the meta-state distance (Eq. (2)) may not be trivial, in terms of both the definition of a proper distance, and the effort required to calculate it (as in Eq. (4)). We can, thus, consider a simplified meta-state distance function, which ignores the system-trajectory distance, and only accounts for the environment distance. Such a simplification means that our meta-state distance function measures distances in a projection of the meta-state space, which abstracts away the system trajectory (simulation output). In practice, we can do so by returning an uninformative, "zero distance" output for every pair of system trajectories.

4.3.2 Simplified sampling of meta-states for planning with RRT. We also recognize that meta-state sampling, as described before for planning with RRT-based node selection, involves sampling of scenes and simulating them, and hence incurs a cost in terms of simulation effort. Further, since these meta-state samples are independent of each other, we cannot benefit from the aforementioned incremental computation, making this cost potentially significant. Nevertheless, as the simplified distance function described above ignores the simulation output anyways, when using it for node selection, there is also no need to waste computational effort on simulation of the sampled scenes. Meaning, in that case, we should coincidentally rely on a simplified meta-state sampling procedure, which returns non-simulated scenes (in our case, by just sampling the track's obstacle collection). Since theses scenes can be viewed as redundant meta-states with a trajectory of length one, they are still in the meta-state space, and this choice can simply be presented as domain-agnostic biasing of the sampling procedure (biasing of the sampling procedure is a common practice in SBMP). With this simplification, simulation effort is only invested in node expansion, and a solution can only be discovered during that phase of the tree growth.

4.4 Node expansion

Consider our meta-control space is based on "replacement" ($\oplus$) mutation operations, i.e., in our running example, removing existing obstacles and adding new ones in their place. As mentioned, in our meta-system, we do not have access to a steering function; meaning, we cannot easily find meta-controls to take our meta-system to an exact destination (e.g., the goal region or a sampled meta-state). Taking notes from the "kinodynamic planning" literature [40, 61, 63, 65], in cases where we only have access to a "forward propagation" model, we may simply consider extension of a random meta-control(s) from the selected node, during node expansion. This assumption should not sacrifice the algorithm's probabilistic completeness [63].

As further recognized by the "kinodynamic planning" literature [63, 65], the step-size for the sampled controls may either be random or predetermined. Yet, unlike kinodynamic planning, where the step-size is determined by a simple scalar expressing time duration (there, Δt), in our

formulation, the step-size corresponds to the more-complex "size of environment-mutation;" we identify two user-controllable parameters, in two axes, which govern this size. The specification of these parameters, to control the step-size, can practically be encoded as a "biased" action sampler.

4.4.1 Environment-mutation width. The first parameter accounts for the number of elements (in the case of our running example—obstacles) we replace in the environment. The user may choose to consider a constant number (e.g., every meta-control always replaces only a single, randomly-chosen obstacle), or randomly choose one (e.g., replace a randomly-selected subset of a random size of obstacles, according to a uniform distribution); for brevity, we refer to the first option as "constant width" and to the latter as "random width." Choosing the latter option makes it possible to occasionally sample a meta-control that replaces all the environment elements in the collection (obstacles). With that, we can essentially start a new search tree from a newly-sampled initial meta-state. This can be effective in improving exploration and can help quickly pull the search tree away from local minima. We may choose to allow such steps only in "exploration iterations" or also when trying to steer towards the goal. Notably, unlike kinodynamic planning, the meta-control sampling is *dependent* on the selected step-size: we should first determine how many obstacles to replace (i.e., the mutation width), and then how to replace them.

4.4.2 Environment-mutation depth: environment-element perturbation. To sample a random meta-control, after choosing the mutation-width n, we shall randomly select a subset of n elements (in our example, obstacles) to replace in the environment, and, finally, randomly select their replacement. The "distance" between the elements (obstacles) to be replaced, and their corresponding replacements, conveys the "mutation depth." Currently, we may assume the selection of the subset of elements for replacement is uniformly random. The simplest way to then determine this replacement, with no need for specialized knowledge, is to uniformly sample a new environment element for each element removed. Yet, allowing each removed element to be replaced with a completely arbitrary new element would mean that the mutation depth can be arbitrarily large. This, in turn, would mean that even if the mutation width is low, the distance between the initial meta-state and the posterior one, after applying the mutation, can also be arbitrarily large, and at an uncontrollable direction. This can hinder our ability to incrementally guide the search-tree growth. A better alternative would be to only locally perturb each element chosen for removal—yet, this is dependent on the availability of such a perturbation procedure. In our example, perturbation of the geometric location of chosen obstacles, e.g., by adding Gaussian white noise to their original location, is easy and trivial; the standard deviation of this Gaussian noise distribution would control the tightness of the perturbation and, by such, the potential mutation depth.

While this will not be evaluated in this paper, given additional, "gray-box" domain-knowledge, we may further inform and bias the meta-control sampling procedure, e.g., using the meta-state distance function calculation, by actively identifying environment mutations that are most likely to challenge the system, or by identifying weak points in the system trajectory.

4.5 On the usage of domain knowledge

In the previous sub-sections, when discussing the planning procedure, we covered several planner parameters the user can specify in order to control and direct the search-tree growth. It is important to differentiate between domain-agnostic planner parameters, and domain-specific knowledge/procedures. Specifically, the specification of the mutation-width sampling distribution, and choice of meta-control sampling technique (uninformed replacement vs. perturbation) during node expansion; the choice of node selection technique (e.g. random, greedy, or RRT); the goal-bias ratio; and choice whether or not to use a simplified distance function (for RRT) are all *not* considered domain-knowledge. Still, to practically enable some of these options, we do

require access to some domain-specific procedures, including an environment-element perturbation procedure, distance-to-failure heuristic, environment distance, and system-trajectory distance.

Our formulation allows us to gradually integrate such domain knowledge to improve the search, based on its availability (as we demonstrate in the experimental results to follow), and can even work with no domain knowledge at all. This comes in contrast to standard techniques covered in Sec. 1.3, which *require* the user to define non-trivially the robustness function, and/or provide hard-to-achieve probabilistic knowledge on prior distributions, and/or have access to the system model. The type of domain knowledge we might consider is mostly heuristic and comparative, which is arguably more intuitive, less-specialized, and easier to obtain. For example, the distance-to-failure heuristic only retrospectively scores a given scenario, to indicate if this system run seems "close" to a failure—it does not require knowledge on how to generate a failure; our environment distance function only indicates how "similar" one environment is to another—it does not require knowledge on the distribution of adversarial environments; our meta-control sampler only requires knowledge on how to locally modify environment elements—it does not require understanding of which elements challenge the system.

Overall, we do not assume any knowledge on the transition dynamics in the meta-space, the system's dynamic model in the environment, or the way the system interacts with the environment. We also, as we recall, make no assumptions about the NN controller or the way it was trained.

5 Experimental evaluation

We tested our falsification approach (summarized in Alg. 1) in the context of our running-example system described in Sec. 2.5: an autonomous car in an "obstructed track" environment, navigating to its end zone, while trying to avoid collision.

5.1 Scenario

To falsify the autonomous car, we were searching for a placement $(x, y) \in \mathbb{R}^2$ for three circular obstacles on a sinuous track, which would cause the car to steer into collision. We considered three scenarios, representing three levels of difficulty of the falsification problem: easy, medium, and hard. The scenarios differ by the length of the track the system must travel on, where a longer track means the obstacles may spread across a larger area; this, in turn, leads to a larger search space and, theoretically, to a harder falsification problem. Simulations of these scenarios were conducted using our lightweight and open-source Python simulation engine, LiteRacer [66]. A screenshot from one of the simulations is provided in Fig. 1.

The track shape is defined as the padded area around the curve $y = 0.8 \cdot \sin(x)$, where the track width set to 1.6 units. For the easy problem, the track is defined in the range $x \in [0, 3\pi]$, for the medium one—$x \in [0, 5\pi]$, and for the hard one—$x \in [0, 7\pi]$. The other scenario properties were set as follows: the car bounding box is of size 0.2×0.4 units, with its frame of origin located in between the back wheels. The initial car position is at $(x = 0, y = 0)$, with the heading set to match the track curve direction, and the steering angle set to 0. The car maximum speed is capped at 0.4 units/second, while the steering speed is limited to $[-10, 10]$ degrees/second, and the steering angle is limited to $[-60, 60]$ degrees. The control frequency is 1Hz. The sensor is placed between the front wheels; the sensor range is 2 units, the sensor angle is in the range $[-72, 72]$ degrees, and the observation image resolution is 100×50 pixels. The obstacles are of radius $r = 0.1$ units.

The controller was trained using Reinforcement Learning (RL), based on numerous randomly-seeded LiteRacer simulations, using OpenAI Gym [67] and the StableBaselines [68] standard implementation of the Soft Actor-Critic (SAC) algorithm, until apparent convergence to the desired behavior. We, nonetheless, treat this controller as a black-box.

5.2 Comparison

To study and prove the benefit of our approach, we compared the computational effort required by different algorithms, both baselines and variations of our algorithm, to falsify the controller, in the each of the described scenarios. In each scenario, for each algorithm, we ran 20 randomly-seeded falsification attempts and measured the average effort invested in each one.

5.2.1 The algorithms in comparison. We compared eight different falsification algorithms: three baselines, and five variants of meta-planning. Most important to notice, each approach requires a different amount of domain knowledge to run. We list ahead the algorithms in comparison, also stating the domain knowledge required by each one. The baselines are as follows: **(1) Uniform environment sampling**, serving as a basic baseline. This algorithm requires no domain knowledge. **(2) A genetic algorithm** (as prescribed in [34]). This algorithm requires access to (i) an environment-perturbation procedure, (ii) an environment-crossover procedure, (iii) distance-to-failure heuristic (used as a fitness function). **(3) A Bayesian-Optimization-based algorithm** (as prescribed in [24]). This algorithm requires access to (i) a distance-to-failure heuristic (used as value function), (ii) optimizer parameters, such as kernels and an acquisition function, (iii) extensive expert knowledge on the scenario, to apply the algorithm to it, as we detail below.

For the genetic algorithm, we used a population of 4 environments in each generation. To evolve the population, we used both crossover operations between environment pairs (random merger of the obstacle sets of two environments), and perturbation-mutation operations for individual environments (as we use for meta-control). In each new generations, two of the samples were generated through crossover, and two through perturbation. In both cases, the selection of environments for mutation relied on a "fitness function" based on the distance-to-failure (as suggested in [34])—effectively acting as a goal bias. The Bayesian-Optimization-based falsification algorithm (and other optimization-based falsifiers), as explained, cannot natively be used in our described scenario, and required several expert-guided adaptations to be applicable. First, we used our testing model, in which the input is the scene (and not a continuous external scalar signal), and the observation signal is generated during the simulation. Second, we used our distance-to-collision heuristic as the optimization objective, instead of the non-trivial robustness. Third, as the number of variables must be predefined, we modified the environment to include three explicit vectors to encode the obstacle locations, instead of using a flexible collection. Fourth, we needed to consider an alternative and less-trivial parametrization for the obstacle location, as the (x, y)-parameterization does not correspond to a continuous box; we incorporated expert knowledge to define the "distance along the track" and the "closeness to the right shoulder" as the alternative parameters. To implement algorithm, we used the standard open-source "Bayesian Optimization" Python package [69]. Finally, the approach requires defining various parameters, such as kernels and an acquisition function; for that, we used the default options suggested by the BO package.

The other five algorithms in comparison are variants of our meta-planning-based falsification algorithm summarized in Alg. 1. The variants differ in the flag setting in the node selection and node expansion procedures and rely on a gradually-increasing amount of domain-knowledge. They are as follows: **(4a) Random search tree with unlimited mutation depth.** This variant requires no domain knowledge. **(4b) Random search tree.** This variant requires access to (i) an environment-perturbation procedure. **(4c) Greedy search tree.** This variant requires access to (i) an environment-perturbation procedure, (ii) a distance-to- failure heuristic. **(4d) RRT with simplified node selection.** This variant requires access to (i) an environment-perturbation procedure, (ii) a distance-to-failure heuristic, (iii) an environment distance. **(4e) RRT.** This variant requires access to (i) an environment-perturbation procedure, (ii) a distance-to-failure heuristic, (iii) an environment distance, (iv) a trajectory distance.

For all variants, we extended a single random meta-control with an unlimited mutation width during node expansion, corresponding to replacement of a random subset of the obstacles collection. For the first variant, the mutation depth was unlimited, meaning, the subset of obstacles could be replaced with uniformly-sampled new obstacles. For all other variants, the mutation depth was limited to a local perturbation, defined by adding white noise (with standard deviation $(2, 2)$) around the original location of each replaced obstacle. For clarity, variants (4a) and (4b) used no goal bias, variant (4c) used a goal bias value of 1, and variants (4d) and (4e) used a goal bias value of 0.8 (i.e., 20% exploration). All eight algorithms require a basic procedure for environment-sampling.

Our implementation of all falsification algorithms is open-sourced and available at [66].

5.2.2 The comparison metrics. We measure and compare the falsification effort using two metrics: the number of environments examined (i.e., tests) until finding a falsifying one, and the total number of control loops simulated, which is an indicator for the total simulation length until solution.

Measuring explicitly the simulation length is important. In other approaches, the simulation length per test is roughly fixed; thus, when comparing those, it is typically sufficient to measure and compare the number of tested environments (i.e., number of simulations), as a proxy for the overall effort. In contrast, one of the novel contributions of this paper is showing that we can reduce the length of simulation needed to test new environments, by exploiting incremental simulation. This means that, with our approach, the simulation length per environment may vary (greatly). It is thus insufficient to merely examine the number of tests when comparing other approaches to ours. We must actually measure and compare the total length of simulation conducted until finding a solution. The number of control loops of the system is simply a proxy for comparing the simulation length—a measure more stable than the implementation-and-hardware-sensitive "clock time."

It is important to keep in mind, when comparing the performance of the algorithms, the amount of knowledge each one requires. We loosely rank the algorithms according to their domain knowledge requirements as follows, from lowest to highest: (1) and (4a) require no domain knowledge, followed closely by (4b), which requires minimal knowledge—only a perturbation procedure. Further up is (4c), as the first to require a goal heuristic, followed closely by (2), which requires also a crossover procedure. Then, we rank (4d) and (4e), which, while giving up on the crossover procedure, require distance measures. Finally, we rank (3), which, while it does not require distances, overall requires the most expert intervention. Surely, given more domain knowledge, we expect an algorithm to perform better. Thus, the following comparison needs to be made in the context of this ranking.

Table 1. Summary of results: average computational effort to find a falsifying environment using different algorithms, in each of the three scenarios. For ease of comparison, results are presented using actual values, and as percentage, in relation to the uniform baseline. Lower is better. The numbers in brackets in the bottom row represent the effort for tree expansion only, not including simulation performed during node selection.

Algorithm Category	Algorithm	Easy scenario				Medium scenario				Hard scenario			
		Average effort				Average effort				Average effort			
		Tests		Control loops		Tests		Control loops		Tests		Control loops	
		#	%	#	%	#	%	#	%	#	%	#	%
Random search	(1) Uniform sampling	84	100%	3009	100%	182	100%	11564	100%	240	100%	21726	100%
Optimization (passive)	(2) Genetic algorithm	96	114%	3457	114%	193	106%	12586	108%	211	87%	19357	89%
Optimization (active)	(3) Bayesian optimization	59	71%	2199	73%	123	68%	7901	68%	156	65%	14411	66%
Meta-planning (ours)	(4a) Random tree with unlimited mutation depth	117	139%	3731	123%	135	74%	7192	62%	195	81%	14547	66%
	(4b) Random tree	112	133%	3354	111%	133	73%	6537	56%	197	82%	12673	58%
	(4c) Greedy tree	78	93%	2509	83%	115	63%	5503	47%	143	59%	10372	47%
	(4d) Simplified RRT	57	68%	1766	58%	96	52%	4703	40%	110	45%	7518	34%
	(4e) RRT	72 60	86% (72%)	2397 (1987)	79% (66%)	88 (74)	48% (41%)	4524 (3642)	39% (31%)	120 (99)	50% (41%)	8250 (6371)	37% (29%)

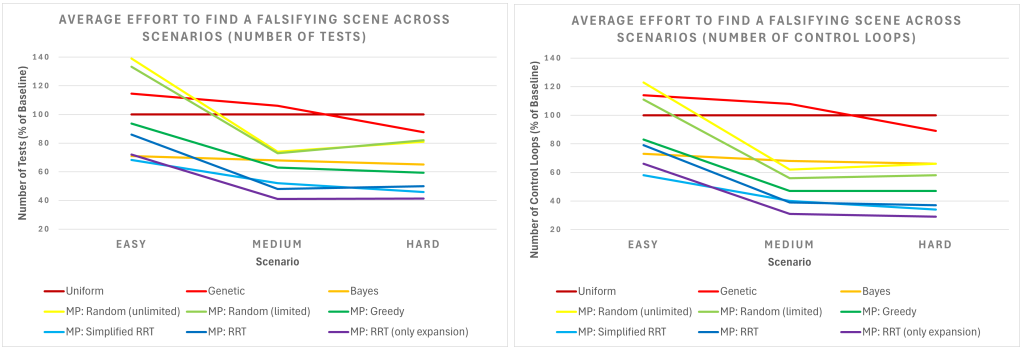


Fig. 6. A chart visualizing the trend in the average effort to find a falsifying scene, across the three scenarios.

5.3 Results

Table 1 presents the average computational effort consumed by each algorithm, when solving each of the falsification problems. Fig. 6 is a graphical representation of these measures, showcasing the trend across the three scenarios. Fig. 7 showcases the performance distribution across the falsification attempts for the medium scenario. Additional theoretical discussion is brought in Appendix B.

From a top view, we can see that the performance ranking of the approaches remains roughly consistent across the three scenarios: the meta-planning approach is the clear winner, followed by the BO algorithm, and then the genetic algorithm. This is especially impressive when taking into account the domain knowledge requirements, as detailed and ranked before. In that regard, examining more specifically the variants of meta-planning, we can see a direct correlation between the amount of domain knowledge incorporated to the performance, with the RRT variants consistently outperforming the greedy variant, which outperforms the random variants.

For all the algorithms, when comparing the results across the three scenarios, we can see a clear trend: as the problem gets harder, the relative improvement in comparison to the uniform baseline increases. In other words, these approaches manage to keep up better than the uniform baseline with the increase in problem complexity, as we would expect. Looking more closely at the first scenario, we see that the least-informed algorithms (the genetic algorithm and the random variants of meta-planning) actually performed worse than the uniform baseline. Since this problem was easy, it seems that the added complexity in these approach was counter-productive. The other algorithms, all of which leveraged a goal heuristic, did manage to achieve (moderate) improvement over the baseline. Examining the harder, second and third scenarios, we can that the uniform sampling approach quickly becomes ineffective; there, even our most basic variant (4a) managed to significantly improve the uniform baseline, in both performance measures. As this variant leverages no domain knowledge, this supports the intuition for the effectiveness of the meta-planning sampling pattern.

All variants of our approach outperformed the genetic algorithm almost across the board, both in terms of the number of environments tested, and the simulation effort. The goal-biased variants of our approach also clearly outperformed the BO algorithm in both performance measures—despite the BO requiring more domain knowledge. Outperforming the baselines in terms of the number of tests means that, even without considering the benefits of incremental simulation, our approach would still come out on top in this comparison. Even our random variants, although outperformed by the BO baseline in terms of the number of tests, still managed, thanks to the incremental simulation, to mostly outperform it in terms of the simulation effort. This is remarkable, given the fact they did not have access to the goal heuristic given to BO.

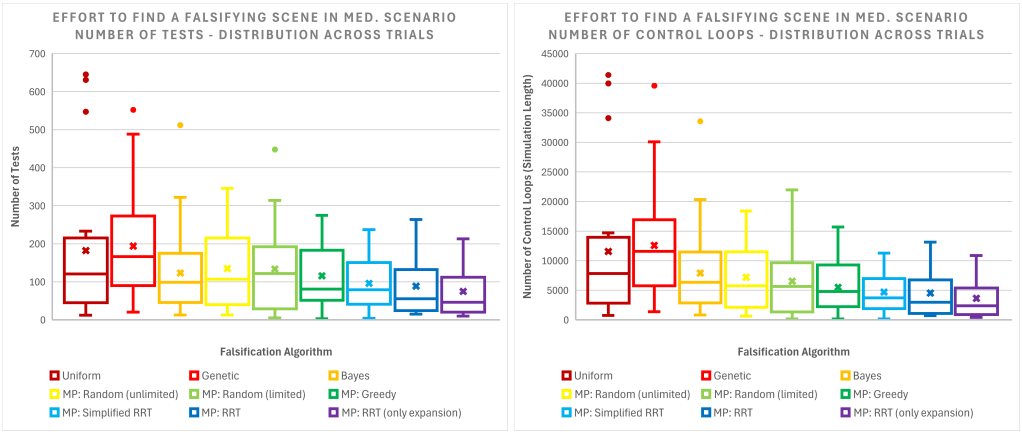


Fig. 7. "Box and whisker charts" representing the distribution of effort across trials, for the medium scenario. Each box represents the two central distribution quantiles; the central line represents the median, and "x"—the mean; the whiskers extend to the maximum and minimum values, excluding "outliers" (marked explicitly).

Further, as the results prove, the effect of exploiting the incremental simulation—one of our novel contributions—is indeed significant. With meta-planning, we can see a great difference when comparing the improvement (against the baseline) in the number of tests and the improvement in the number of control loops: in the latter, the improvement is greater. This difference indicates the additional and significant contribution the simulation reuse had in this case. If it had no effect, we would expect the improvement in both measures to be similar (as is the case with the baselines).

Finally, we also observe that using the (unsimplified) RRT variant (4e) did not lead to practical benefits in performance over the simplified counterpart (4d). With RRT, we recall that some of the cost includes explicit meta-state sampling (i.e., simulation) during the node selection process. This extra simulation cost for node selection invested by RRT (which was not required for the simplified variant) counterbalanced the benefit to the search, which led to an overall similar performance. Nevertheless, to compare the search effectiveness, we measured separately the effort invested by RRT in tree expansion (numbers in brackets); in this comparison, it becomes clear that RRT outperforms the simplified variant. This supports the claim that searching the meta-state space, rather than searching the input space (as the simplified variant does), holds the potential to better guide the search for a falsifying witness.

6 Conclusion

This paper investigated the problem of efficiently finding falsifying inputs to an autonomous system guided by a black-box controller under general specifications. We first identified that, for such systems that observe their environments using a high-dimensional sensor, the input should correspond to a description of an environment (used to initiate a simulation), rather than a sensor signal directly, as most often considered by existing approaches. We formulated accordingly a more appropriate testing model, which inputted the static environment description, and outputted the system trajectory and observation history. Considering this model, we suggested a novel reformulation of the falsification problem as planning a trajectory for a meta-system, or, in short, meta-planning. Meta-states of this meta-system encapsulate input-output pairs (i.e., fully-simulated scenes), and transitioning between them is done by applying meta-controls—mutations to the input, which invoke an update to the output. As we showed, such an update could be performed incrementally in each expansion iteration of the planning algorithm—only performing a partial scene-simulation,

starting from the time-step in which the observation-history is compromised. The meta-planning problem can be solved with off-the-shelf planners, and we specifically provided guidelines on how to employ the sampling-based RRT algorithm for that task. We finished with experimental evaluation of the approach for the problem of an autonomous car with a NN-controller, trained to lead it along a track, while avoiding collision with randomly-scattered obstacles. Our experiments proved that meta-planning could significantly reduce the falsification effort, in terms of both the number of samples (tests), and the simulation effort per sample—even over approaches informed by more extensive domain knowledge.

With that said, we would like to make a final note: by no means do we try to claim here that meta-planning will always outperform any other falsification approach. Falsification is a process based on randomness. From our experience, different approaches may be more effective in different scenarios. What we do prove clearly in this paper is that meta-planning is a worthy addition to the portfolio of available techniques.

References

[1] Seshia, S.A., Sadigh, D., Sastry, S.S.: Toward verified artificial intelligence. Commun. ACM 65(7), 46–55 (Jun 2022)

[2] Singh, B., Kumar, R., Singh, V.P.: Reinforcement learning in robotic applications: A comprehensive survey. Artif. Intell. Rev. 55(2), 945–990 (Feb 2022)

[3] Zare, M., Kebria, P.M., Khosravi, A., Nahavandi, S.: A Survey of Imitation Learning: Algorithms, Recent Developments, and Challenges. IEEE Transactions on Cybernetics pp. 1–14 (2024)

[4] Deshmukh, J.V., Sankaranarayanan, S.: Formal Techniques for Verification and Testing of Cyber-Physical Systems. In: Al Faruque, M.A., Canedo, A. (eds.) Design Automation of Cyber-Physical Systems, pp. 69–105. Springer International Publishing, Cham (2019)

[5] Lee, E.A., Varaiya, P.: Structure and Interpretation of Signals and Systems. LeeVaraiya.org, second edn. (2011)

[6] Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (Sfcs 1977). pp. 46–57 (Oct 1977)

[7] Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 97–117. Springer International Publishing, Cham (2017)

[8] Ehlers, R.: Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks (Aug 2017)

[9] Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient Formal Safety Analysis of Neural Networks. In: Advances in Neural Information Processing Systems. vol. 31. Curran Associates, Inc. (2018)

[10] Weng, L., Zhang, H., Chen, H., Song, Z., Hsieh, C.J., Daniel, L., Boning, D., Dhillon, I.: Towards Fast Computation of Certified Robustness for ReLU Networks. In: Proceedings of the 35th International Conference on Machine Learning. pp. 5276–5285. PMLR (Jul 2018)

[11] Bansal, S., Chen, M., Herbert, S., Tomlin, C.J.: Hamilton-Jacobi reachability: A brief overview and recent advances. In: 2017 IEEE 56th Annual Conference on Decision and Control (CDC). pp. 2242–2253 (Dec 2017)

[12] Mitchell, I., Bayen, A., Tomlin, C.: A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games. IEEE Transactions on Automatic Control 50(7), 947–957 (Jul 2005)

[13] Akintunde, M.E., Lomuscio, A., Maganti, L., Pirovano, E.: Reachability Analysis for Neural Agent-Environment Systems. In: Proceedings of the Sixteenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2018) (2018)

[14] Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: Verifying safety properties of hybrid systems with neural network controllers. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. pp. 169–178. HSCC '19, Association for Computing Machinery, New York, NY, USA (Apr 2019)

[15] Ivanov, R., Carpenter, T.J., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verifying the Safety of Autonomous Systems with Neural Network Controllers. ACM Trans. Embed. Comput. Syst. 20(1), 7:1–7:26 (Dec 2020)

[16] Xiang, W., Johnson, T.T.: Reachability Analysis and Safety Verification for Neural Network Control Systems (May 2018)

[17] Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's Decidable about Hybrid Automata? Journal of Computer and System Sciences 57(1), 94–124 (Aug 1998)

[18] Corso, A., Moss, R., Koren, M., Lee, R., Kochenderfer, M.: A Survey of Algorithms for Black-Box Safety Validation of Cyber-Physical Systems. Journal of Artificial Intelligence Research 72, 377–428 (Oct 2021)

[19] Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoretical Computer Science 410(42), 4262–4291 (Sep 2009)

[20] Akazaki, T., Hasuo, I.: Time Robustness in MTL and Expressivity in Hybrid System Falsification. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 356–374. Springer International Publishing, Cham (2015)
[21] Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems 2(4), 255–299 (Nov 1990)
[22] Maler, O., Nickovic, D.: Monitoring Temporal Properties of Continuous Signals. In: Lakhnech, Y., Yovine, S. (eds.) Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems. pp. 152–166. Springer, Berlin, Heidelberg (2004)
[23] Ramezani, Z.: On Optimization-Based Falsification of Cyber Physical Systems. Ph.D. thesis, Chalmers University of Technology, Göteborg (2022)
[24] Deshmukh, J., Horvat, M., Jin, X., Majumdar, R., Prabhu, V.S.: Testing Cyber-Physical Systems through Bayesian Optimization. ACM Trans. Embed. Comput. Syst. 16(5s), 170:1–170:18 (Sep 2017)
[25] Ghosh, S., Berkenkamp, F., Ranade, G., Qadeer, S., Kapoor, A.: Verifying Controllers Against Adversarial Examples with Bayesian Optimization. In: 2018 IEEE International Conference on Robotics and Automation (ICRA). pp. 7306–7313 (May 2018)
[26] Mathesen, L., Pedrielli, G., Fainekos, G.: Efficient Optimization-Based Falsification of Cyber-Physical Systems with Multiple Conjunctive Requirements. In: IEEE International Conference on Automation Science and Engineering (CASE). pp. 732–737 (Aug 2021)
[27] Zhang, Z., Hasuo, I., Arcaini, P.: Multi-armed Bandits for Boolean Connectives in Hybrid System Falsification. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 401–420. Springer International Publishing, Cham (2019)
[28] Ramezani, Z., Šehić, K., Nardi, L., Åkesson, K.: Falsification of Cyber-Physical Systems using Bayesian Optimization (Feb 2023)
[29] Ramezani, Z., Claessen, K., Smallbone, N., Fabian, M., Åkesson, K.: Testing Cyber–Physical Systems Using a Line-Search Falsification Method. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 41(8), 2393–2406 (Aug 2022)
[30] Abbas, H., Fainekos, G., Sankaranarayanan, S., Ivančić, F., Gupta, A.: Probabilistic Temporal Logic Falsification of Cyber-Physical Systems. ACM Trans. Embed. Comput. Syst. 12(2s), 95:1–95:30 (May 2013)
[31] Aerts, A., Tong Minh, B., Mousavi, M.R., Reniers, M.A.: Temporal Logic Falsification of Cyber-Physical Systems: An Input-Signal-Space Optimization Approach. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 214–223 (Apr 2018)
[32] Sankaranarayanan, S., Fainekos, G.: Falsification of temporal properties of hybrid systems using the cross-entropy method. In: Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control. pp. 125–134. HSCC '12, Association for Computing Machinery, New York, NY, USA (Apr 2012)
[33] Kim, Y., Kochenderfer, M.J.: Improving Aircraft Collision Risk Estimation Using the Cross-Entropy Method. Journal of Air Transportation 24(2), 55–62 (Apr 2016)
[34] Zhao, Q., Krogh, B., Hubbard, P.: Generating test inputs for embedded control systems. IEEE Control Systems Magazine 23(4), 49–57 (Aug 2003)
[35] Hansen, N., Ostermeier, A.: Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In: Proceedings of IEEE International Conference on Evolutionary Computation. pp. 312–317 (May 1996)
[36] Li, G., Li, Y., Jha, S., Tsai, T., Sullivan, M., Hari, S.K.S., Kalbarczyk, Z., Iyer, R.: AV-FUZZER: Finding Safety Violations in Autonomous Driving Systems. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE). pp. 25–36 (Oct 2020)
[37] Zhang, Z., Lyu, D., Arcaini, P., Ma, L., Hasuo, I., Zhao, J.: FalsifAI: Falsification of AI-Enabled Hybrid Control Systems Guided by Time-Aware Coverage Criteria. IEEE Transactions on Software Engineering 49(4), 1842–1859 (Apr 2023)
[38] Dreossi, T., Donzé, A., Seshia, S.A.: Compositional Falsification of Cyber-Physical Systems with Machine Learning Components. J. Autom. Reason. 63(4), 1031–1053 (Dec 2019)
[39] Orthey, A., Chamzas, C., Kavraki, L.E.: Sampling-Based Motion Planning: A Comparative Review. Annual Review of Control, Robotics, and Autonomous Systems 7(Volume 7, 2024), 285–310 (Jul 2024)
[40] Elbanhawi, M., Simic, M.: Sampling-Based Robot Motion Planning: A Review. IEEE Access 2, 56–77 (2014)
[41] Cheng, P., Kumar, V.: Sampling-based Falsification and Verification of Controllers for Continuous Dynamic Systems. The International Journal of Robotics Research (Nov 2008)
[42] Aineto, D., Scala, E., Onaindia, E., Serina, I.: Falsification of Cyber-Physical Systems Using PDDL+ Planning. Proceedings of the International Conference on Automated Planning and Scheduling 33, 2–6 (Jul 2023)
[43] Plaku, E., Kavraki, L.E., Vardi, M.Y.: Hybrid systems: From verification to falsification by combining motion planning and discrete search. Formal Methods in System Design 34(2), 157–182 (Apr 2009)
[44] Plaku, E., Kavraki, L.E., Vardi, M.Y.: Falsification of LTL safety properties in hybrid systems. International Journal on Software Tools for Technology Transfer 15(4), 305–320 (Aug 2013)

[45] Esposito, J.M., Kim, J., Kumar, V.: Adaptive RRTs for Validating Hybrid Robotic Control Systems. In: Erdmann, M., Overmars, M., Hsu, D., van der Stappen, F. (eds.) Algorithmic Foundations of Robotics VI, pp. 107–121. Springer, Berlin, Heidelberg (2005)
[46] Tuncali, C.E., Fainekos, G.: Rapidly-exploring Random Trees for Testing Automated Vehicles. In: 2019 IEEE Intelligent Transportation Systems Conference (ITSC). pp. 661–666 (Oct 2019)
[47] Ernst, G., Sedwards, S., Zhang, Z., Hasuo, I.: Falsification of Hybrid Systems Using Adaptive Probabilistic Search. ACM Trans. Model. Comput. Simul. 31(3), 18:1–18:22 (Jul 2021)
[48] Lamport, L.: Proving the Correctness of Multiprocess Programs. IEEE Transactions on Software Engineering SE-3(2), 125–143 (Mar 1977)
[49] Fox, M., Long, D.: PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. Journal of Artificial Intelligence Research 20, 61–124 (Dec 2003)
[50] De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence. pp. 854–860. IJCAI '13, AAAI Press, Beijing, China (Aug 2013)
[51] Zhang, Z., Ernst, G., Sedwards, S., Arcaini, P., Hasuo, I.: Two-Layered Falsification of Hybrid Systems Guided by Monte Carlo Tree Search. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37(11), 2894–2905 (Nov 2018)
[52] Julian, K.D., Lee, R., Kochenderfer, M.J.: Validation of image-based neural network controllers through adaptive stress testing. In: 2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC). pp. 1–7. IEEE (2020)
[53] Yamagata, Y., Liu, S., Akazaki, T., Duan, Y., Hao, J.: Falsification of Cyber-Physical Systems Using Deep Reinforcement Learning. IEEE Transactions on Software Engineering 47(12), 2823–2840 (Dec 2021)
[54] Ernst, G., Arcaini, P., Bennani, I., Chandratre, A., Donzé, A., Fainekos, G., Frehse, G., Gaaloul, K., Inoue, J., Khandait, T., Mathesen, L., Menghi, C., Pedrielli, G., Pouzet, M., Waga, M., Yaghoubi, S., Yamagata, Y., Zhang, Z.: ARCH-COMP 2021 Category Report: Falsification with Validation of Results. In: 8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21). pp. 133–112 (2021)
[55] Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 254–257. Springer, Berlin, Heidelberg (2011)
[56] Donzé, A.: Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. pp. 167–170. Springer, Berlin, Heidelberg (2010)
[57] Jin, X., Deshmukh, J.V., Kapinski, J., Ueda, K., Butts, K.: Powertrain control verification benchmark. In: Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control. pp. 253–262. HSCC '14, Association for Computing Machinery, New York, NY, USA (Apr 2014)
[58] Sankaranarayanan, S., Kumar, S.A., Cameron, F., Bequette, B.W., Fainekos, G., Maahs, D.M.: Model-based falsification of an artificial pancreas control system. SIGBED Rev. 14(2), 24–33 (Mar 2017)
[59] Dreossi, T., Fremont, D.J., Ghosh, S., Kim, E., Ravanbakhsh, H., Vazquez-Chanlatte, M., Seshia, S.A.: VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 432–442. Springer International Publishing, Cham (2019)
[60] Fremont, D.J., Kim, E., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: A language for scenario specification and data generation. Machine Learning 112(10), 3805–3849 (Oct 2023)
[61] Li, Y., Littlefield, Z., Bekris, K.E.: Asymptotically optimal sampling-based kinodynamic planning. The International Journal of Robotics Research 35(5), 528–564 (Apr 2016)
[62] Cortés, J., Siméon, T.: Sampling-Based Tree Planners (RRT, EST, and Variations). In: Ang, M.H., Khatib, O., Siciliano, B. (eds.) Encyclopedia of Robotics, pp. 1–9. Springer, Berlin, Heidelberg (2020)
[63] LaValle, S.M., Kuffner, J.J.: Randomized Kinodynamic Planning. The International Journal of Robotics Research 20(5), 378–400 (May 2001)
[64] Hsu, D., Kindel, R., Latombe, J.C., Rock, S.: Randomized Kinodynamic Motion Planning with Moving Obstacles. The International Journal of Robotics Research 21(3), 233–255 (Mar 2002)
[65] Sucan, I.A., Kavraki, L.E.: A Sampling-Based Tree Planner for Systems With Complex Dynamics. IEEE Transactions on Robotics 28(1), 116–131 (Feb 2012)
[66] Elimelech, K., Lahijanian, M., Vardi, M.Y., Kavraki, L.E.: LiteRacer: A lightweight autonomous vehicle simulator for benchmarking and development of formal verification techniques. In: Workshop on Software Challenges in Formal Methods for Robotics (FMR), in Conjunction with ICRA 2024. Yokohama, Japan (May 2024)
[67] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: OpenAI Gym (Jun 2016)
[68] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., Dormann, N.: Stable-Baselines3: Reliable Reinforcement Learning Implementations. Journal of Machine Learning Research 22(268), 1–8 (2021)
[69] Nogueira, F.: Bayesian Optimization: Open source constrained global optimization tool for Python (2014/)

A Algorithms

```
Algorithm falsification_as_meta_planning()
    // A DOMAIN-INDEPENDENT PROCEDURE
    tree ← []
    random_meta_state ← sample_meta_state()
    root_node ← Node(meta_state ← random_meta_state, parent ← Null)
    tree.add(root_node)
    while not timeout do
        selection_result ← select_node(tree)
        if selection_result is MetaState then
            Return selection_result                    // GOAL FOUND WHILE SAMPLING FOR NODE SELECTION!
        else
            selected_node ← selection_result
            expansion_result ← selected_node.expand_node(tree,selected_node)
            if expansion_result is MetaState then
                Return expansion_result                // GOAL FOUND DURING NODE EXPANSION!
    Return Null                                        // FAILED TO FIND FALSIFYING EXAMPLE—PERHAPS RESTART PROCESS

Procedure sample_meta_state()
    // A DOMAIN-INDEPENDENT PROCEDURE
    random_env ← sample_env()
    initial_scene ← Scene(random_env, INITIAL_STATE)
    trajectory, history ← simulate_system(initial_scene)
    Return (random_env, trajectory, history)

Procedure sample_env()
    // A DOMAIN-SPECIFIC PROCEDURE
    env ← initialize environment of type "track"
    set track parameters
    for i from 1 to number of obstacles do
        (x, y, r) ← sample a random obstacle position
        env.obstacles.add((x, y, r))
    Return env
```

Algorithm 1: Meta-planning-based falsification using sampling-based forward search.

```
Procedure select_node(tree)
    // A DOMAIN-INDEPENDENT PROCEDURE
    if RANDOM_NODE_SELECTION then
        selected_node ← randomly selected node from tree
    else if GREEDY_NODE_SELECTION then
        selected_node ← argmax_{node in tree} distance_to_failure(node.meta_state)
    else if RRT_NODE_SELECTION then
        r ← random number in the range (0, 1)
        if r < GOAL_BIAS then
            selected_node ← argmax_{node in tree} distance_to_failure(node.meta_state)
        else
            if STANDARD_DISTANCE then
                random_meta_state ← sample_meta_state()
                if status(random_meta_state) = 0 then
                    Return random_meta_state           // GOAL FOUND WHILE SAMPLING FOR NODE SELECTION!
                else
                    selected_node ←
                      argmax_{node in tree} meta_state_distance(node.meta_state, random_meta_state)
            else if SIMPLIFIED_DISTANCE then
                random_env ← sample_env()
                selected_node ←
                  argmax_{node in tree} env_distance(node.meta_state.env, random_meta_state.env)
    Return selected_node
```

Algorithm 2: Node selection.

```
Procedure expand_node(tree, node)
    // A DOMAIN-INDEPENDENT PROCEDURE
    for i from 1 to EXPANSION_BREADTH do
        new_meta_state ← random_meta_control(node.meta_state)
        new_node ← Node(meta_state ← new_meta_state, parent ← node)
        tree.add(new_node)
        if status(new_meta_state) = 0 then
            Return new_meta_state                          // GOAL FOUND DURING NODE EXPANSION!
    Return Null                                            // GOAL NOT YET FOUND

Procedure random_meta_control(meta_state)
    // A DOMAIN-INDEPENDENT PROCEDURE (META-DYNAMICS)
    mutated_env ← random_replacement_mutation(meta_state.env)          // MUTATE ENV
                          // CALC TIMESTAMP IN WHICH MUTATION STARTS AFFECTING OBSERVATION HISTORY
    history_compromise_timestamp ← find_history_compromise_timestamp(meta_state, mutated_env)
    history_comporomise_state ← meta_state.traj[history_compromise_timestamp]
            // SIMULATE STARTING FROM TIMESTAMP, AND INCREMENTALLY UPDATE ORIGINAL TRAJECTORY/HISTORY
    initial_scene ← Scene(mutated_env, history_comporomise_state)
    trajectory_suffix, history_suffix ← simulate_system(initial_scene)
    updated_traj ← [meta_state.traj[0 : history_compromise_timestamp], trajectory_suffix]
    updated_history ← [meta_state.history[0 : history_compromise_timestamp], history_suffix]
    Return (mutated_env, updated_traj, updated_history)          // RETURN UPDATED META-STATE

Procedure random_replacement_mutation(env)
    // A DOMAIN-SPECIFIC PROCEDURE (THE ⊕ OPERATOR)
    mutated_env ← copy(env)
    if CONST_MUTATION_WIDTH then                                  // DETERMINE ENV MUTATION SIZE
        n ← STEP_SIZE
    else if RANDOM_MUTATION_WIDTH then
        n ← randomly selected number from 1 to len(mutated_env.obstacles) of obstacles to replace
                                                                  // PERFORM RANDOM ENV MUTATION
    obstacles_to_replace ← randomly selected n obstacles from mutated_env.obstacles
    for obstacle in obstacles_to_replace do
        mutated_env.obstacles.remove(obstacle)
        if UNLIMITED_MUTATION_DEPTH then
            new_obstacle ← uniformly sample a random obstacle position
        else if LIMITED_MUTATION_DEPTH then
            new_x ← obstacle.x+ randomly sampled scalar noise
            new_y ← obstacle.y+ randomly sampled scalar noise
            new_r ← obstacle.r+ randomly sampled scalar noise
            new_obstacle ← (new_x, new_y, new_r)
        mutated_env.obstacles.add(new_obstacle)
    Return mutated_env

Procedure find_history_compromise_timestamp(meta_state, new_env)
    // A DOMAIN-INDEPENDENT VERSION
    for i from 0 to length(meta_state.traj) − 1 do
        new_z ← h(meta_state.traj[i], new_env)                    // SIMULATE OBSERVATION IN NEW ENV
        if not new_z = meta_state.observation_history[i] then
            Return i                                              // OBSERVATION INVALID IN NEW ENV
    Return length(meta_state.traj)                                // HISTORY VALID IN NEW ENV

Procedure find_history_compromise_timestamp(meta_state, new_env)
    // AN EFFICIENT DOMAIN-SPECIFIC VERSION: AVOIDING SIMULATION USING KNOWLEDGE ON SENSOR MODEL
    for i from 0 to length(meta_state.traj) − 1 do
        observed_area ← estimate observed portion of track in meta_state.observation_history[i] based on sensor
          properties
        if any of added/removed obstacles overlap with observed_area then
            Return i                                              // OBSERVATION INVALID IN NEW ENV
    Return length(meta_state.traj)                                // HISTORY VALID IN NEW ENV
```

Algorithm 3: Node expansion.

B Extended theoretical discussion

For the interested reader, we provide an extended theoretical discussion, justifying our novel approach and clarifying the differences from existing approaches and our advantages over them.

B.1 Falsification via meta-planning vs. optimization

Meta-planning and most optimization-based-falsification approaches are, theoretically, applicable in similar contexts, as they both consider the sampled inputs to be independent of the simulation. Though, we recall that those approaches were developed considering a more limited testing model (predefined number of input variables, input variables in a continuous box, etc.). To compare those approaches to ours, we must first adapt them to our more generic testing model (as we did in our experimental results).

Beyond that, the two have some major differences in the way they solve the falsification problem, which we would like to highlight. Falsification, at the most basic level, is the problem of searching for an example of a simulation, a proof, in the space of simulations, that demonstrates a certain behavior. A simulation consists of (i) the definition of the scenario used to set it up—this is the input in the testing model we defined; and (ii) the resulting system run—this is the output in the testing model we defined. In other words, the meta-state we defined represents a simulation, and the meta-state-space is the space of simulations. **Meta-planning simply means leveraging a planning algorithm as a way to search the space of simulations, for its inherent ability to balance space exploration and goal bias.** We lay out four important points regarding this statement, to explain the difference between meta-planning and the existing optimization-based approaches, and the inherent advantages meta-planning holds.

First, regarding the computational problem. Falsification is a search problem. Search is a different computational problem than optimization. Existing approaches clearly try to fit this search problem into an optimization form, while converting the specification into a quantitative measure. This, while often is indeed natural, may sometimes be simply inappropriate. For example, when considering a purely qualitative specification, such as that the system trajectory must be from a discrete set of trajectories, be from a certain homotopy class, or hold some abstract property expressing a human perference. Additionally, most optimization approaches specifically look at the signal robustness as the quantitative objective, which is yet another limitation, as it may not always be available/well-defined/suitable for the problem. Our approach maintains the original definition of the problem as a search problem, whose solution can still be, though less restrictively, guided by a quantitative heuristic. This allows our approach to potentially fit a wider set of scenarios.

Second, regarding the state space. The existing approaches search the space of inputs, i.e., a lower dimensional projection of the simulation space. While they surely leverage the output signal to guide the search in that space (e.g., by estimating an input-to-robustness model, or by setting preference for the next region to explore), this still marginalizes potentially useful information. This information can be used, e.g., to ensure appropriate coverage of the search space. Without it, we might be fooled to think that by choosing different inputs we are exploring well the space of simulations, not knowing that all our inputs actually result in the same trajectory. Since meta-planning searches the simulation space, our approach inherently holds this theoretical advantage.

Third, regarding the search approach. We note that all search approaches, including both existing optimization-based approaches and our approach, rely on gradually selecting a sequence of simulations to be tested, until one of these tests complies with the desired condition; at that point, all the futile tests are discarded, and the final one is returned. This sequential nature is not unique to our approach. The differences between the approaches are in (i) the technique used to

determine the order of test cases, and, (ii) the relation between the test cases. Our approach is unique in both of these aspects. In the context of (i), existing optimization approaches generate a linear sequence of tests—at each point in the search only one test (or a few, in case of a genetic algorithm) is "active" and, based on its result, a new test case is generated. In our approach, we more generally build a *tree* of tests, in which numerous test remain "active" and can be used as a starting point for selecting the next test case (as illustrated in Fig. 8). Maintaining this tree of tests can practically allow us to backtrack during the search, to revisit past samples, if those seemed more promising than the latest one. This is a major advantage over the optimization approaches, which can easily be stuck in a local optima or dead-end, without the ability to backtrack. Even though some modern approaches add policies to perturb the model when it seems to be stuck in a local minima, the problem still inherently exists, as former samples are discarded. Explicitly maintaining all past samples also allows us to leverage space-coverage heuristics, to make sure the space is properly explored, and the samples do not "collapse" into a certain region. In the context of (ii), in the existing approaches, every new simulation is independent from the ones before it and has to be tested anew. In our approach, every new test case is explicitly generated from one of the previous test cases—meaning that the sequence of test cases represents a connected trajectory. This, as we explained, is a significant advantage, allowing us to efficiently update the simulation of each new test case, rather than calculating it from scratch.

Fourth, regarding the resulting sampling pattern. The incrementality of test-case generation in our approach, together with the choice to grow a tree of samples, results in a sampling pattern that densely covers whole regions of the search space. Since the search is generally not fully-greedy, this tree can grow in various directions. Further, allowing for mutations of high-width, as we mentioned, also allows us to "re-root the tree," essentially growing multiple of such dense regions in different parts of the space, ensuring proper coverage and exploration. Overall, this sampling scheme steadily and effectively "combs" through the environment space until finding a falsifying example. This pattern seems to be very effective in challenging scenarios like the one in our running example, when the goals are sparse (as illustrated in Fig. 8). In that scenario, the environment-state specified the location for multiple obstacles, and the car only failed when these obstacles were in specific and highly-correlated configurations. Also, small changes in the location of obstacles could cause discrete changes in the homotopy class of the car trajectory, resulting in large changes in its distance-to-failure. This overall meant that the search goal corresponded to many small and sparsely-scattered goal meta-regions, and that the optimization field was non-smooth and non-convex, with many local minima. In these cases, standard sequential-sampling approaches, which result in a more sparse pattern, may struggle to to "hit" a small goal region.

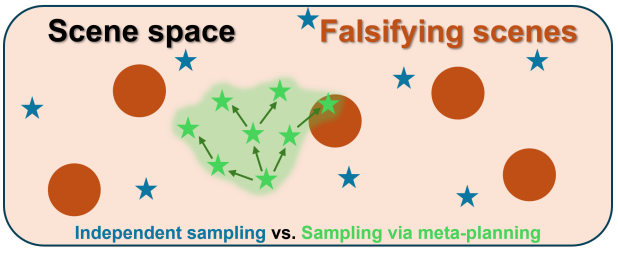


Fig. 8. Illustration of the scene-sampling patterns of our incremental, tree-based, meta-planning sampler (in green), and a standard, sequential sampler (in blue). In this challenging case, where the search goals are sparse, our meta-planning approach seem to require less samples until "hitting" one of the goal regions.

Let us provide some additional insight regarding the specific optimization algorithms used in our experimental evaluation.

Falsification via meta-planning vs. BO. In our experiments, meta-planning outperformed BO. In addition to the points above, it is possible that the complex relationship between the input and output (scene and trajectory) in our scenario particularly debilitated the ability of the BO-baseline to learn a useful model to guide its sampling. Another point that may have caused this struggle is the arbitrary fixation of the obstacles into specific variables ("obstacle 1/2/3"), which does not capture the inherent symmetry under permutation of the obstacle collection. Our approach manages to avoid this pitfall by relying on flexible element-collections to define the input environments.

Falsification via meta-planning vs. genetic algorithms. In our experiments, meta-planning outperformed the genetic-algorithm. While both meta-planning and the genetic algorithm we examined utilized mutation operations for sample generation, our experiments proved that meta-planning was a more effective search approach. Unlike the genetic algorithm, we are not forced to perform serial "blanket" updates to the entire sample population, at each iteration. We used evolutionary operations to grow a planning-tree of samples, by dynamically selecting and evolving a single promising sample at a time. By such, we are also not restricted to a sample population of a predefined size, as the number of leaves in the planning tree can dynamically grow, according to the number of promising search directions discovered. We may also note that the crossover operation used by the genetic algorithm was probably ineffective in our example, as combining two "halves" of close-to-failure environments does not necessarily result in a close-to-failure environment. Furthermore, one may even view this genetic algorithm as a particular, restricted solution algorithm to our formulated meta-planning problem. To support this argument, it is easy to see that the behavior of a standard evolutionary algorithm can be achieved with a redundant meta-planning forward-search algorithm, by first extending an initial set of samples from a root node, and then continuing to grow the tree using a Breadth-First-Search (BFS) strategy. More advanced variations can be demonstrated as well, by appropriately managing the node priority-queue.

B.1.1 Drawbacks. While meta-planning is not guaranteed to always outperform existing falsifiers, for the high generality of its formulation, we cannot point out an inherent theoretical drawback of meta-planning over optimization-based approaches. Perhaps a possible difficulty one may experience in the meta-planning approach is the need for the user to define the planner properties, such as the mutation operation, mutation depth and width, and node selection technique. However, these choices, we believe, are not more demanding than setting up the parameters of any optimizer, and are arguably more intuitive to define than, e.g., a sampling kernel for Bayesian optimization.

B.2 Falsification via meta-planning vs. planning

Clearly, both our meta-planning approach and the "planning-based-falsification" approaches covered in Sec. 1.3 utilize motion planning algorithms for falsification, for their ability to effectively search a high-dimensional space. Yet, both the application-contexts and the manner-of-usage of these two approaches are completely different, making them not direct competitors of each other.

"Planning" approaches search the system-state space and build a planning-tree in that space, where each node represents a system state and each branch a (prefix of) a system trajectory. Extension of a new node from one of the leaves is done by incrementally simulating the system from the respective state for a short time frame, while injecting some "intervention," and then marking the end state as a new node. In contrast, meta-planning builds a planning tree in the meta-state space (i.e., simulation space), where each tree node encapsulates a scene and *a full system-trajectory*, from the initial state to termination. Extension of a new node from one of

the leaves is done by taking and modifying the respective scene and re-simulating the system (from start to finish) in it—then combining the two into a new tree node, extended from that leaf. While we do leverage incremental simulation, this is only a computational "trick," to avoid redundant re-calculation of overlapping segments of the trajectory; each node still encapsulate an full simulation.

In "planning," the algorithm terminates when reaching an "unsafe" system state. This reflects the fact such algorithms can only be used to falsify safety specifications, which are compromised by a single state—unlike us, who can handle both safety and liveness specifications. In that case, the planning algorithm returns the system trajectory given by the tree branch, from the root to that unsafe state, as the witness of failure, alongside the sequence of interventions that were injected along this path. In meta-planning, the algorithm will terminate when the simulation corresponding to a tree leaf we extended indicates system failure; in that case, the meta-planning algorithm returns that simulation (meta-state), representing both the entire failing-trajectory and the failure-causing scene; the rest of meta-state-trajectory, leading to that leaf, is not of interest, as it represents a sequence of non-failing simulations, only standing for the history of the search process.

To state it explicitly, planning-based-falsification and meta-planning do not share the same use case. Specifically, "planning" is not applicable in the context of our running example. "Planning" approaches are only useful for falsification problems that look for a falsifying sequence of run-time interventions (like for a *controlled* system), and not for a passive initial scenario (as we do here); without injection of intervention, there is no way for them to build a search tree. While one may perhaps consider the obstacles as injectable interventions, this would lead to a plethora of new questions regarding if and when and where to place the obstacles in coordination with those segments of simulation. Answering such questioning requires a serious investigation, which is surely beyond the scope of this paper.