Metamorphic Relations via Relaxations: An Approach to Obtain Oracles for Action-Policy Testing[*]

Hasan F. Eniser,[1] Timo P. Gros,[2] Valentin Wüstholz,[3] Jörg Hoffmann,[2,4] Maria Christakis[1]

[1] MPI-SWS, Kaiserslautern and Saarbrücken, Germany
{hfeniser, maria}@mpi-sws.org
[2] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
⟨lastname⟩@cs.uni-saarland.de
[3] ConsenSys, Kaiserslautern, Germany
wuestholz@gmail.com
[4] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany

Abstract

Testing is a promising way to gain trust in a learned action policy π, in particular if π is a neural network. A "bug" in this context is undesirable or fatal policy behavior, e.g. satisfying a failure condition. But how do we distinguish whether such behavior is due to bad policy decisions, or is actually unavoidable under the given circumstances? This requires knowledge about optimal solutions, which defeats the scalability of testing. Related problems occur in software testing when the correct program output is not known. Metamorphic testing addresses this through metamorphic relations, specifying how a given change to the input should affect the output, thus providing an oracle for the correct output. Yet how to obtain such metamorphic relations for action policies? Here we show that the well-explored concept of relaxations can serve that purpose. If state s' is a relaxation of state s, and π fails on s' but does not fail on s, then we know that π contains a bug manifested on s'. We contribute an initial exploration of this idea, in the context of failure testing of neural network policies π learned by RL in simulated environments. We design fuzzing strategies for test-case generation, and metamorphic oracles leveraging simple manually designed relaxations. In experiments on three single-agent games, our technology is able to identify true bugs – avoidable failures of π – quite effectively.

1 Introduction

Action policies represented by neural networks (NN) are highly successful in complex sequential decision making problems, in particular in games (Mnih et al. 2015; Silver et al. 2016, 2018), and increasingly in AI Planning (Issakkimuthu, Fern, and Tadepalli 2018; Groshev et al. 2018; Garg, Bajpai, and Mausam 2019; Toyer et al. 2020; Karia and Srivastava 2021). Once a policy π has been learned, it can be used to make real-time decisions in dynamic environments, simply by calling $\pi(s)$ on the current state s to obtain the next action. This approach, however, comes with obvious safety concerns due to potential policy bugs, i.e., undesirable or fatal policy behavior. Testing is a natural paradigm, given its scalability, to address this.

But what is a "bug" in this context? In many environments, undesirable or fatal behavior can be unavoidable – e.g. traffic making it impossible to avoid a crash in autonomous driving, or a state in which it is impossible for a bipedal robot to keep its balance. Such situations are not bugs in π, as the bad behavior is not actually due to bad policy decisions. In general, in order to know whether or not a situation constitutes a bug in π, we need to know what the optimal policies are. This defeats the scalability of testing.

Prior work on testing in sequential decision making does not address this issue. This work considers a "system" that takes decisions in an environment, and tries to find situations where a failure condition ϕ is satisfied (e.g. (Dreossi et al. 2015; Akazaki et al. 2018; Koren et al. 2018; Ernst et al. 2019; Lee et al. 2020), see (Corso et al. 2021) for a recent overview). This implicitly assumes that a correctly designed system – in our case, a learned action policy – can avoid ϕ.

Steinmetz et al. (2021) recently pointed out this shortcoming, and analyzed the possibility to identify sub-optimal policy behavior through upper- and lower-bounding techniques. Here, we take inspiration from software testing instead, providing an alternative technique to identify avoidable failures.

Not knowing the optimal policies is akin to not knowing the correct output of a program. Metamorphic testing (Chen, Cheung, and Yiu 1998) addresses the latter by testing program behavior on inputs chosen such that it is known how the respective outputs should relate. That is, one specifies a *metamorphic relation*, encompassing a relation R^I over inputs together with a corresponding necessary relation R^O over outputs. If, for inputs i, i' with $R^I(i, i')$, the outputs o, o' do not satisfy $R^O(o, o')$, then we know there is a bug. Thus R^O provides a test *oracle* for the correct output.

That oracle, however, still requires knowledge about correct outputs, in the form of R^O. For example, action decisions in autonomous driving can be tested using metamorphic relations derived from well-known human-designed

[*]A slightly extended version of this paper will be published at the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22).

rules, such as "speed down by 25% if rainy" (Zhang et al. 2018; Tian et al. 2018; Deng et al. 2020). But how to come by metamorphic relations in general planning?

Here we answer that question in terms of a relation R^O not over specific output actions a vs. a' of a policy, but over *the space of solutions below states s vs. s'*. Such relations are very common and well explored in AI, namely in the form of over-approximations obtained through *relaxation*. Assume s and s' are related in terms of a relaxation relation $R(s, s')$ identifying that s' is easier to solve than s. If π fails on s' but not on s, then we know that π contains a bug manifested on s'. This is our key insight: *relaxations provide a means to specify metamorphic relations and thus a test oracle in general planning*. Note that, furthermore, this approach captures sequential policy behavior, rather than merely immediate outputs as in all other works on metamorphic testing.

As indicated in our notation above, in this paper we focus on *state relaxations* R, that modify only the state, not any other aspects of the agent's task. Such relaxations are often quite natural and easy to obtain. When obstacles need to be avoided, states can be relaxed by removing obstacles; in planning with resources, relaxations can increase resource availability; in planning under time constraints, those constraints can be relaxed (e.g. postponing a deadline).

We do not yet investigate the automatic generation of such relaxations. Instead, we run case studies in three 2D-world single-agent games involving (fixed or moving) obstacles, and manually design relaxation relations $R(s, s')$ where s' has easier-to-avoid obstacles. Importantly, while this involves manual per-domain labor, it requires hardly any domain *knowledge* (relaxing obstacles is trivial) – in contrast to knowing the difference between the optimal solutions for s and s', which is the kind of knowledge we would need to design a traditional metamorphic output relation R^O.

Nevertheless, the automated design of relaxations for this purpose, including ones going beyond state relaxations, remains of course an important topic for future work. For this, there is huge a potential to draw on the literature on relaxation design for heuristic functions (e.g. (Bonet and Geffner 2001; Edelkamp 2001; Helmert et al. 2014; Domshlak, Hoffmann, and Katz 2015)). This is non-trivial though, for a variety of reasons. In particular, rather than relaxing radically to obtain an efficiently computable heuristic function, we need to relax cautiously, in small steps, to be able to identify bugs.

Our testing framework addresses MDPs, of which we require access only to a simulator (given state s and action a, output an outcome state s'). For test-state generation, we take inspiration from fuzzing (Miller, Fredriksen, and So 1990), which mutates program inputs randomly with a bias to maximize diversity. We transfer this idea to our setting by taking input mutations to be random action applications, and measuring test-state diversity in terms of Euclidean distance. We implemented all our techniques in a framework we call π-fuzz. We evaluate π-fuzz on three single-agent games, with policies learned by RL. Our experiments show that fuzzing is effective in generating a diverse set of states, and that our metamorphic oracles are able to identify thousands of unique bugs even in well-trained policies.[1]

2 Context and Notations

Our methods address discrete-time Markov decision processes, as follows. An MDP is a tuple $M = (S, A, T, S_0)$ of **states** S; **actions** A; **transition probability function** $T : S \times A \mapsto \mathcal{D}(S)$ where $\mathcal{D}(S)$ denotes the set of probability distributions over S; and **initial states** $S_0 \subseteq S$ (of which one $s_0 \in S_0$ will be chosen randomly at execution time).

A **policy**, also **agent**, is a function $\pi : S \mapsto A$ which chooses actions in S. We consider policies π represented by neural networks (NN). The policy is typically trained to maximize rewards associated with states or state transitions. Our approach is agnostic to how this is done. A **run** of a policy π on an (arbitrary) state $s_0 \in S$ is a state/action sequence $\sigma = \langle s_0, a_0, s_1, a_1, \dots \rangle$ where, for all i, $a_i = \pi(s_i)$ and $\mu(s_{i+1}) > 0$ where $\mu = T(s_i, a_i)$.

Note that this definition of policy is restricted in terms of being memoryless and deterministic. Both restrictions can be lifted in principle, but deterministic memoryless policies are relevant in their own right and form a natural starting point for the investigation of metamorphic action-policy testing. We assume that π is represented as an NN classifier whose final layer can also be interpreted as a probability distribution $\hat{\pi}(s) \in \mathcal{D}(A)$ over actions. We make use of the latter in fuzzing, by sampling $\hat{\pi}(s)$ in order to explore MDPs in which random actions do not lead to interesting states.

We assume a given non-temporal **failure condition** ϕ that should be avoided by the agent (exploring our approach for temporal ϕ remains a topic for future work). We say that a run σ **fails** if there exists s_i along σ such that $s_i \models \phi$; otherwise, we say that σ **succeeds**. We denote by $P_\phi(\pi, s)$ the probability that the run of π on s fails, and by $P^*_\phi(s)$ the minimal such probability achieved by any policy.

We do not assume that we have a declarative model of M; a simulator suffices to apply our methods. We merely assume that the representation of states is state-variable based, i.e., each s is uniquely identified by a value assignment to a vector of **state variables** $(v_1, \dots, v_n)$. The domains of the state variables do not matter to our approach, so long as Euclidean distance can be defined (needed in our coverage notions). For simplicity, in this paper we assume that the state variables are real-valued, i.e., states s map each v_i to $\mathbb{R}$.

We refer to the simulator as the **environment**, denoted E. It provides the programmatic interfaces E.**rndInit**() which returns a random initial state $s_0 \in S_0$; E.**setState**(s) which sets the environment state to $s \in S$; and E.**step**(s, a) which, given $s \in S$ and $a \in A$, picks a state s' according to the distribution $T(s, a)$, and outputs s'.

We furthermore assume that E has a parameter ρ – the random seed – and an interface E.**setSeed**(r) setting $\rho := r$. We use that interface in part of our methodology to fix specific environment behaviors and identify bugs pertaining to those. Namely, whenever we check whether π contains a bug manifested below a state s, directly before running π on s we call E.setSeed(r), determinizing T as a function of

[1]The source code of π-fuzz and of all our experiments will be made publicly available upon publication.

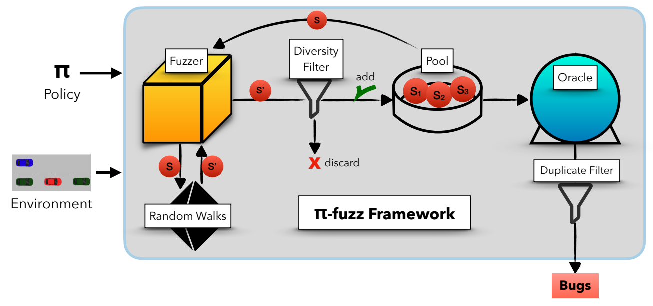


Figure 1: Overview of π-fuzz framework.

state, action, and run length so far. We denote the resulting unique **run of** π **on** s **given random seed** r by $E^r.\sigma[\pi, s]$.

3 π-fuzz Policy Fuzzing Framework

Figure 1 provides a high-level overview of our π-fuzz policy-testing framework. π-fuzz takes as input a policy under test π and an environment E. The framework consists of two main components: the **fuzzer**, which generates a diverse pool of test states s_i; and the **oracle**, which identifies policy bugs among these test states. The fuzzer uses random walks to generate new states, which are then filtered by diversity to obtain the pool. We will describe our fuzzing algorithm in detail in Section 5. Our key contribution is the design of the oracle, via metamorphic relations based on relaxations. A duplicate filter at the end of this pipeline serves to provide unique bugs as the output of π-fuzz.

π-fuzz is implemented as a generic policy tester, independent of any specific environment (in particular, we use the same π-fuzz implementation across our case studies in this paper). The input interface for π-fuzz consists of the neural network representation of π, in the PyTorch format; the aforementioned programmatic interface of E implementing $E.\text{rndInit}()$, $E.\text{setState}(s)$, $E.\text{step}(s, a)$and $E.\text{setSeed}(r)$; as well as a programmatic interface for metamorphic operations underlying the oracle, explained next.

4 Metamorphic Oracles via Relaxation

We define the notions of policy bug we use in our context. We spell out the principle of relaxation-based metamorphic oracles; then specify the oracle we use in our case studies; then discuss how suitable relaxations may be obtained in general, in particular considering the relaxation literature.

4.1 Policy Bugs: Definition

We consider two notions of "bugs", one of which is specific to a fixed environment behavior, while the other quantifies over all possible such behaviors.

Definition 1 (Bug)**.** Let $M = (S, A, T)$ be an MDP, E a simulator for M, π a policy, and $s \in S$ a state.

(i) We say that s is a **bug** in π if $P_\phi(\pi, s) > P^*_\phi(s)$.

(ii) Given a random seed $\rho = r$, we say that the run $E^r.\sigma[\pi, s]$ is a **seed-bug** in π if $E^r.\sigma[\pi, s]$ fails, but there exists a policy π' s.t. $E^r.\sigma[\pi', s]$ succeeds.

Bugs (i) arguably capture the canonical understanding of policy bugs when testing failure-avoidance ability in a probabilistic environment. Seed-bugs (ii) are an approximation that allows to consider individual environment behaviors. If π fails but π' succeeds given the same fixed random seed, then this indicates that π is faulty. This is, however, not necessarily the case: (ii) does *not* in general imply (i), because the decisions causing failure on r may be beneficial on other environment behaviors. Hence (ii) merely is a pragmatical proxy for (i). That said, on deterministic environments (i) and (ii) coincide; and in our case studies, most states s with seed-bugs found by our metamorphic oracle are in fact bugs. Also, (ii) is much faster to evaluate than (i), which makes it useful for practical debugging purposes.

Seed-bugs are best characterized by the actual run $E^r.\sigma[\pi, s]$, rather than the state s alone, as there can be many different environment behaviors below s and only some of them may exhibit the observed failure.

4.2 Metamorphic Oracles: The Principle

Obviously, Definition 1 cannot be tested efficiently on large state spaces. We adapt the idea of metamorphic testing to solve this issue. The key element is a state relaxation:

Definition 2 (State Relaxation)**.** Let $M = (S, A, T)$ be an MDP, and E a simulator for M. We say that $t \in S$ **relaxes** $s \in S$ if for every policy π_s there exists a policy π_t such that, for every random-seed $\rho = r$, whenever $E^r.\sigma[\pi_s, s]$ succeeds then $E^r.\sigma[\pi_t, t]$ succeeds as well.

We say that $R \subseteq S \times S$ is a (state) **relaxation** if, for every $(s, t) \in R$, t relaxes s.

We will illustrate and discuss this definition below. In a nutshell, a relaxed state t allows to adapt any policy for s to achieve the same (or more) failure-avoidance ability. In that sense, intuitively, "t is easier to solve than s". Definition 2 captures the most general condition under which this is the case, and where our metamorphic oracles hence work as intended.

Namely, the idea is quite simple. If t is easier to solve than s, but the policy π is worse on t than on s, then π's behavior on t must be wrong:

Proposition 3 (Metamorphic Oracle)**.** Let $M = (S, A, T)$ be an MDP, E a simulator for M, and R a relaxation. Let $s, t \in S$ be states s.t. $(s, t) \in R$. We have:

(i) If $P_\phi(\pi, s) < P_\phi(\pi, t)$, then t is a bug in π.

(ii) If $E^r.\sigma[\pi, s]$ succeeds but $E^r.\sigma[\pi, t]$ fails, then $E^r.\sigma[\pi, t]$ is a seed-bug in π.

Proof. (i): We have $P^*_\phi(s) \geq P^*_\phi(t)$ by the definition of relaxations. Hence we get $P_\phi(\pi, t) > P_\phi(\pi, s) \geq P^*_\phi(s) \geq P^*_\phi(t)$ which shows the claim.

(ii): As the run of π on s succeeds and $(s, t) \in R$, by the definition of relaxations there exists a policy π' for which the run on t succeeds. As the latter is not the case for π, $E^r.\sigma[\pi, t]$ is a seed-bug in π. □

For illustration of the kind of relaxations we employ here, let us briefly consider the case studies we contribute. We experiment with stochastic games where an agent moves in a

2D-world and needs to reach a target position while avoiding (fixed or moving) obstacles. Our relaxation relations R modify the game landscape by removing obstacles, or moving them in a way that makes them easier to avoid.

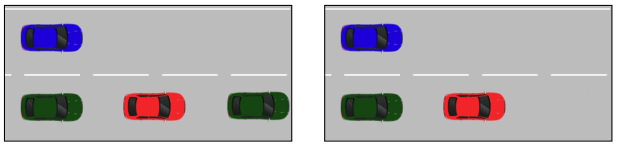


Figure 2: Illustration of relaxations as per Definition 2 in our Highway case study.

Figure 2 illustrates this in our **Highway** case study, which involves a car (red) navigating traffic (blue and green) on a 2-lane highway. Less traffic is easier to navigate. In the sense of Definition 2, whenever π_s manages to avoid crashing into traffic when started from s, we can achieve the same when starting from t simply by taking the same driving decisions; i.e., the desired policy π_t behaves like π_s on the respective corresponding states. If the policy π under test takes different decisions on t, which crash more frequently, then π exhibits a bug on t.

A remarkable special case of Definition 2 is that of deterministic transitions, where the run of a policy from a state is unique, and t relaxes s if and only if either t is solvable (a succeeding policy exists), or s is unsolvable; in particular, all solvable states relax each other. This is very generous from a definitorial point of view, but it still makes perfect sense for bug detection: if s is solvable and $R(s,t)$, then we know that t is solvable. This is precisely the most general condition under which we can detect avoidable failures by comparing the behavior of π across states s, t: if $R(s,t)$ and π succeeds on s, then t is solvable, so failure of π on t constitutes a bug. Practical relaxation methods will, of course, instantiate the broad frame of Definition 2 with much more restrictive relations over states, like the relaxations in our case studies.

4.3 Metamorphic Oracles in our Case Studies

Proposition 3 provides a tool to detect bugs and seed bugs, given a state relaxation R as per Definition 2. We turn this into practical oracles for π-fuzz by sampling R a given number of times. Algorithm 1 specifies three different oracles along these lines. Let's consider these from top to bottom.

The BUGORACLE algorithm checks whether a given test-pool state s_i generated by π-fuzz can be identified to be a bug. It does so by comparing $P_\phi(\pi, s_i)$ with $P_\phi(\pi, t_i)$ for *unrelaxed* states t_i, i.e., harder states where $R(t_i, s_i)$. By Proposition 3, if the Boolean return value is 1 then s_i is a bug in π. Evaluating P_ϕ here is a sub-problem, solving which exactly is intractable in itself in large state spaces. In our implementation, we approximate P_ϕ by sample runs.

The BASICSEEDBUGORACLE algorithm proceeds in a similar manner, but checks for seed bugs instead. The random seed r is an input to the oracle (set by the fuzzer, see Section 5) as the oracles's job is to identify bugs given a fixed environment behavior. The oracle returns 1 if π fails on

Algorithm 1: Metamorphic Oracles

```
Function BugOracle ($R, \pi, s_i$):
    evaluate $P_\phi(\pi, s_i)$;
    repeat ORACLE_BUDGET times
        $t_i$ = RANDOMSTATE($\{t_i \mid R(t_i, s_i)\}$);
        evaluate $P_\phi(\pi, t_i)$;
        if $P_\phi(\pi, t_i) < P_\phi(\pi, s_i)$ then
            return 1
    return 0;
Function BasicSeedBugOracle ($R, \pi, s_i, r$):
    if $E^r.\sigma[\pi, s_i]$ fails then
        repeat ORACLE_BUDGET times
            $t_i$ = RANDOMSTATE($\{t_i \mid R(t_i, s_i)\}$);
            if $E^r.\sigma[\pi, t_i]$ succeeds then
                return 1
    return 0;
Function ExtSeedBugOracle ($R, \pi, s_i, r$):
    $B = \{\}$;
    if $E^r.\sigma[\pi, s_i]$ succeeds then
        repeat ORACLE_BUDGET times
            $t_i$ = RANDOMSTATE($\{t_i \mid R(s_i, t_i)\}$);
            if $E^r.\sigma[\pi, t_i]$ fails then
                $B = B \cup \{E^r.\sigma[\pi, t_i]\}$;
    return $B$;
```

the test state s_i but succeeds on one of the unrelaxed states t_i. By Proposition 3, $E^r.\sigma[\pi, s_i]$ is a seed-bug in this case.

EXTSEEDBUGORACLE, finally, is an extension that applies in case π succeeds on the test state s_i given r. In this case, $E^r.\sigma[\pi, s_i]$ cannot be a seed-bug, but we may be able to identify relaxed states t_i as seed-bugs instead. The oracle leverages this possibility in the obvious manner. In our case studies, many additional seed-bugs are found in this way.[2]

To sample the relaxation relation R, our implementation in π-fuzz assumes that R is given in the form of a set of **metamorphic operations**: state-modification operators that either relax the given state (e.g., by removing obstacles) or unrelax it (e.g., by adding obstacles). The sampling then simply consists in applying a randomly chosen metamorphic operation, with randomly chosen parameters (e.g. which obstacles to remove, where to add new obstacles).

Importantly, the magnitude of metamorphic operations affects oracle efficacy. If π works well on s_i and t_i is much easier, it is unlikely that the policy is bad on t_i, thus not leading to the detection of a bug. If π is bad on s_i and t_i is much harder than s_i, it is unlikely that the policy works well on t_i, again not leading to the detection of a bug. Therefore, in both directions, metamorphic operations should be applied cautiously, making small modifications only. Our operations modifying individual state attributes naturally support this.

[2]One can define a similar extended version of BUGORACLE. Such an oracle would be very slow however.

Also, for that reason, we do not chain over metamorphic operations, always applying only a single such operation when sampling R in Algorithm 1.

Section 6 outlines the metamorphic operations we use in each of our case studies. The algorithm parameter `ORACLE_BUDGET` is set to 500 in all our experiments.

4.4 Discussion: How To Obtain Relaxations?

How can state relaxations for metamorphic oracles be obtained? Arguably, relaxations as above – modifying individual state attributes – are easy to come by in many cases, based on trivial domain-specific knowledge. Wherever obstacle avoidance plays a role, the methods from our case studies can be used. Other simple examples include resource consumption (increase availability), and deadlines (postpone) or more generally time windows (broaden). Indeed, the latter three kinds of relaxations can potentially be automated in a domain-independent manner.

It remains of course an important question whether and how we can tap into the potential of existing research on relaxations as underly the computation of heuristic functions (e.g. (Bonet and Geffner 2001; Edelkamp 2001; Helmert et al. 2014; Domshlak, Hoffmann, and Katz 2015)). In this context, note the following differences between our methodology vs. commonly used relaxation methods:

(a) Relaxations normally modify the problem as a whole, not only the state as in Definition 2.

(b) Practical oracles require cautious relaxation in small steps, as outlined above.

In principle, neither of these differences is intrinsic in our framework, i.e., standard relaxations could be plugged in directly. But both (a) and (b) are serious challenges, because the design of heuristic functions has very different requirements. In contrast to (b), heuristic functions must be efficiently computable, which entails strong problem simplifications. Regarding (a), if the relaxation differs radically from the original problem, then care needs to be taken that the learned policy actually generalizes to the relaxed problem. These challenges are not insurmountable (e.g. many known relaxations come with fine-grained refinement operations), but certainly constitute difficult research questions.

Another, perhaps more promising, source of state relaxations can be simulation relations (Milner 1971; Gentilini, Piazza, and Policriti 2003), where $R(s, t)$ holds – t simulates s – iff for every outgoing transition of s there is a corresponding outgoing transition in t, leading to simulating outcome states $R(s', t')$. Such relations can be extracted automatically from standard planning models (Torralba and Hoffmann 2015).

5 Fuzzing Algorithm

We now discuss the fuzzer component from Figure 1 in more detail. The fuzzer builds up a pool of diverse states by relying on two sub-components, namely random walks and diversity analysis. Consider the pseudo-code in Algorithm 2.

First, the fuzzer adds a random initial state to the pool of states P (line 3). Until the fuzzer is interrupted (e.g., via a user-provided time limit), it tries to incrementally expand P.

Algorithm 2: Fuzzing procedure

```
1  Function FUZZER (Env $E$, Policy $\pi$):
2      $P$ = [];
3      $P$ = ADD($E$.RNDINIT(), $P$);
4      while ¬INTERRUPTED() do
5          if RANDOMBOOLEAN(INC_PROB) then
6              $s$ = RANDOMSTATE($P$);
7          else
8              $s$ = $E$.RNDINIT();
9          $s'$ = RNDWALK($E$, $\pi$, $s$);
10         if $\min_{t \in P} d^{\text{Eucl}}(s', t) >$ DIV_THRESH then
11             $P$ = ADD($s'$, $P$);
12     for each $s_i \in P$ do
13         Run oracle on $s_i$ (picking $r$ if needed);
14 Function RNDWALK (Env $E$, Policy $\pi$, State $s$):
15     $E$.SETSTATE($s$);
16     $k$ = RANDOMINTRANGE(0, WALK_LENGTH);
17     if RANDOMBOOLEAN(POL_PROB) then
18         repeat $k$ times
19             $a$ = RANDOMACTION($E$.actions);
20             $s$ = $E$.STEP($s$, $a$);
21     else
22         repeat $k$ times
23             $a$ = RANDOMPOLICYACTION($\hat{\pi}$(s));
24             $s$ = $E$.STEP($s$, $a$);
25     return $s$;
```

To do so, it randomly decides (biased by a probability provided in parameter `INC_PROB` on line 5) to either select a random state from the pool (line 6) or select a new random initial state (line 8). A random walk is then conducted on the resulting state s to obtain a new candidate state s' (line 9). If s' is sufficiently diverse, it is added to P (lines 10 – 11). Here, $d^{\text{Eucl}}(s, s')$ is the Euclidean distance between s and s', and `DIV_THRESH` sets a threshold for the minimum distance to the states already in the pool P.

Once the pool P is final, an oracle is called on each state $s_i \in P$ (lines 12 – 13). If the oracle requires a fixed random seed – like the two seed-bug oracles from Algorithm 1 – then the fuzzer chooses that seed here.[3]

The diversity filter serves (similarly as in software testing) for higher confidence in π's ability to avoid failure, based on broad tests. Our criterion is inspired by recent work on testing neural network classifiers, applying Euclidean distance between activation vectors to a given network layer (Odena et al. 2019). Here we apply this criterion to states – i.e., the NN input vectors – instead.

The RNDWALK procedure conducts random walks in the usual manner, with one noteworthy design decision. Rather

[3]To check for bugs on several different environment behaviors, the oracle would need to be called with several different seeds. Here we show that, in several interesting case studies, many seed-bugs can be found even when trying only a single seed for each s_i.

than always choosing actions uniformly at random (line 19), the algorithm sometimes samples the policy under test instead (line 23; recall that $\hat{\pi}(s) \in \mathcal{D}(A)$ interprets the final layer of the NN policy as a probability distribution over actions). The parameter POL_PROB (line 17) controls the trade-off between these two choices. Sampling the policy makes sense when random actions do not tend to lead to interesting states, e.g. because states quickly become unsolvable. Indeed, as our empirical results show, this method yields strong advantages in one of our case studies.

Regarding the parameters of Algorithm 2, in preliminary experiments we found that INC_PROB $= 0.8$ tends to work well across all of our case studies (for smaller values, exploration is insufficient), so we fix that parameter value. Similarly, we fix POL_PROB $= 0.2$ (for larger values, exploration is insufficient). For DIV_THRESH and WALK_LENGTH, good values depend on domain-specific aspects: typical scale of state diversity, typical run length, typical level of risk incurred by long random walks. We hence fix specific values for each domain, listed as part of our case study descriptions in the next section. For POL_PROB and DIV_THRESH, interesting algorithm performance differences arise from setting these to 0 vs. > 0, so we evaluate these settings in our experiments below.

6 Case Studies

We apply our π-fuzz framework to three case studies, called Highway, LunarLander, and BipedalWalker. Illustrations of their environments are shown in Figure 3. LunarLander and BipedalWalker are popular Gym (Brockman et al. 2016) environments specialized for continuous control. We developed Highway as a new benchmark that simulates a simplified autonomous-driving task, navigating a highway through speed changes and lane changes in a way that avoids collisions with traffic. In all these case studies, the failure condition ϕ is given in terms of a specific environment state in which the agent ends up when it crashes into an obstacle. All agents were trained on a Debian 10 machine with 768 GB of memory, 32 CPUs (Intel(R) Xeon(R) Gold 6134M), and 2 GPUs (V100 Nvidia Tesla with 32 GB of memory).

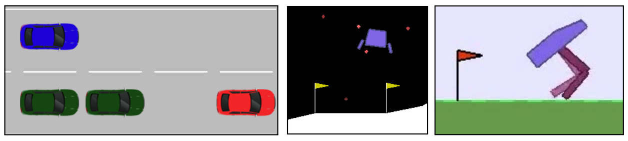


Figure 3: Illustrations of domains used in our evaluation: (a) Highway, (b) LunarLander, (c) BipedalWalker.

We next describe each case study, including the domain itself, the algorithms and parameteres we used for learning the policy π, the metamorphic operations for the oracle, and the domain-specific settings of DIV_THRESH and WALK_LENGTH.

6.1 Highway

The Highway domain consists of a two-lane, finite-length street. The left lane is for *speed maniacs*, who are relatively fast, and the right lane is for *safety freaks*, who are slow. Neither of these actors may change lanes, and their speed is constant. The agent appears at the beginning of the street, and the task is to reach the end without crashing into other cars. There are five discrete actions: switch lane to right or left, speed up, slow down, noop. Other cars may enter or leave the highway stochastically while the agent is moving. In case of a crash, the game ends immediately with a reward of -100 points. Reaching the end of the street is rewarded with $+100$ points. The discount factor is $\gamma = 0.95$, incentivizing the agent to drive fast. Given the road length, the best-case achievable reward is ca. 30.

Policy training. We train our agent using our own implementation of DQN (Mnih et al. 2015). Our agent is well trained, and achieves an average reward of 21 points (after 20000 training episodes).

Metamorphic operations. Relaxed states are generated by removing a random car ahead of the agent, thus reducing the chance of crashing. Conversely, unrelaxed states are generated by adding a car at a safe distance from the agent (not leading to unavoidable crashes).

DIV_THRESH and WALK_LENGTH. We set DIV_THRESH to 3.6 to capture the typical scale of diversity in this domain, which we gauged by inspecting the visual differences between states. We set WALK_LENGTH to 3 to balance the typical risk of random walks, which quickly lead to crashes in this domain.

6.2 LunarLander

The LunarLander domain consists of an uneven lunar surface and a lander with two legs. The lander appears at the top of the environment with a random velocity vector, and the task is to land it on its legs – if the body touches the surface, the lander crashes. There are four discrete actions: firing the bottom engine, the left-hand side engine, the right-hand side engine, noop. The effect of firing an engine is stochastic, following a probability distribution over the force yielded. Touching a leg to the ground yields reward $+100$, touching the body to the ground yields reward -100. There is no discount factor (runs are finite, indeed short, in this game anyway), so the best-case reward is 200.

Policy training. We train our agent using the PPO algorithm (Schulman et al. 2017) implemented in the SB3 library (Raffin et al. 2019). Our agent is well trained, and achieves an average reward of 175 points (after 1 million training episodes).

Metamorphic operations. Relaxed states are generated by decreasing the height of the surface, giving the lander more time to land. Conversely, we generate unrelaxed states by increasing the surface height up to a safe distance.

DIV_THRESH and WALK_LENGTH. We set DIV_THRESH to 0.6 and WALK_LENGTH to 25.

6.3 BipedalWalker

In the BipedalWalker domain, a bipedal robot moves along a finite-length terrain that has a rough surface. The robot's task is to move forward until the end of the terrain. The action space is continuous, with actions being defined by a 4-tuple of numbers $x_i \in [0.0, 1.0]$. Each x_i specifies the force applied to one of the joints of the robot. The actions are deterministic; the only stochastic element in this domain are the terrain (surface) shape and the initial forces in the robot's joints. The best-case achievable reward is $+300$ collected when reaching the end of the terrain, plus small positive rewards that can be collected beforehand. If the robot falls, it receives -100 points, and the game ends immediately. There is no discount factor (as, again, game runs are short).

Policy training. We use the PPO algorithm from SB3 for training. Our agent achieves an average reward of 302 points (after 1 million training episodes).

Metamorphic operations. As smooth surfaces are easier to navigate for the walker, relaxed states are generated by making the terrain smoother, whereas unrelaxed states are generated by making the terrain rougher.

`DIV_THRESH` and `WALK_LENGTH`. We set `DIV_THRESH` to 2.0 and `WALK_LENGTH` to 25.

7 Experiments

Our primary evaluation concerns bug-finding capability, i.e., the number of (seed-)bugs correctly identified by different oracles. We furthermore analyze the impact of the `POL_PROB` and `DIV_THRESH` algorithm parameters. In what follows, we first introduce the oracles we compare, then focus on these evaluations in turn.

We run each experiment 8 times and report statistics over these 8 runs below. Each run was done on a Debian 10 machine with a time limit of 24 hours, 1.5 TB of memory, and 48 CPUs (Intel(R) Xeon(R) CPU E7-8857 v2).

7.1 Competing Oracles

To provide a comprehensive evaluation, we consider not only our metamorphic oracles, but eight oracles in total:

MMBug, MMSeedBugBasic, MMSeedBugExt The oracles from Algorithm 1.

FailureSeedBug This oracle simply flags s_i as a bug iff $E^r.\sigma[\pi, s_i]$ fails.

This is a trivial baseline that does not check avoidability.

RuleSeedBug Prior work on testing of autonomous driving decisions (Zhang et al. 2018; Tian et al. 2018; Deng et al. 2020) suggests a rule-based approach, leveraging readily available human knowledge in the form of driving rules. While such knowledge is not available in our general setting here, to provide at least some comparison to rule-based approaches, the RuleSeedBug oracle explores a simple "don't change decision" rule. It reports pool state s_i as a seed-bug if there is an unrelaxed state t_i, $R(t_i, s_i)$, such that $E^r.\sigma[\pi, t_i]$ succeeds and $\pi(s_i) \neq \pi(t_i)$. The rationale is that what works for t_i also works for s_i so the policy should not change.

As $\pi(s_i) = \pi(t_i)$ is possible but not necessary, this oracle may incorrectly classify s_i as a seed-bug. Here we report only the true positives, measuring the oracle's ability to identify true bugs (which as we shall see is lacking).

PerfectBug, PerfectSeedBug These oracles provide exact measuring lines. Computing them is feasible only for Highway, so we report data only in that case study.

MMSeedBug2Bug Calls MMSeedBugBasic first. If that flags s_i as a bug due to unrelaxed state t_i, checks whether $P_\phi(\pi, t_i) < P_\phi(\pi, s_i)$ and flags s_i as a bug if that is so.

Such seed-bug filtering speeds up bug-finding as we will see. Further, this oracle evaluates how many seed-bugs found by MMSeedBugBasic correspond to bugs.

In MMBug and MMSeedBug2Bug, we evaluate P^ϕ by running the policy 50 times. (Based on limited experiments, this is reasonable in our use cases; using statistical methods to compute P^ϕ up to a confidence bound is future work.)

Prior to considering the empirical data for these oracles, note the following guaranteed relations between the sets of states (or state/seed pairs) they identify as bugs:

RuleSeedBug $\subseteq$ MMSeedBugBasic The true positives of RuleSeedBug are dominated by those of MMSeedBugBasic, because if $R(t_i, s_i)$ and $E^r.\sigma[\pi, t_i]$ succeeds, then s_i is a bug iff $E^r.\sigma[\pi, s_i]$ fails – which is precisely what MMSeedBugBasic is checking.

MMSeedBugBasic $\subseteq$ PerfectSeedBug, MMBug $\subseteq$ PerfectBug By Proposition 3.

PerfectSeedBug $\subseteq$ FailureSeedBug FailureSeedBug catches all seed-bugs but may incorrectly flag non-bugs.

MMSeedBug2Bug $\subseteq$ MMSeedBugBasic, MMSeedBug2Bug $\subseteq$ MMBug By construction.

MMSeedBug2Bug $=$ MMSeedBugBasic $=$ MMBug on deterministic domains where policy runs are unique.

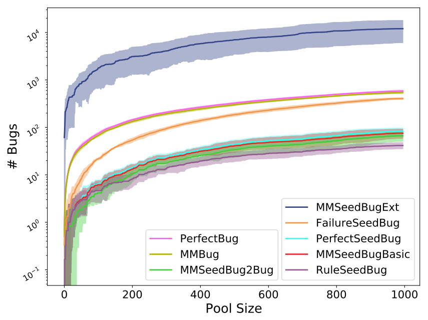
The MMSeedBugExt oracle is incomparable to the others as it is the only one that attempts to find additional (seed-)bugs, beyond the pool states s_i.

7.2 Results: Oracle Capability

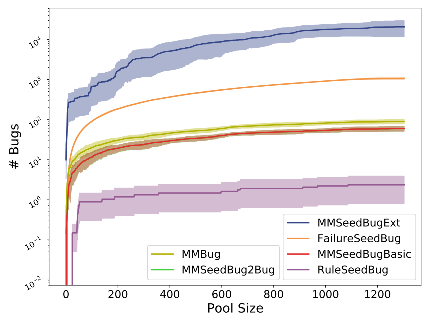
Figure 4 shows our evaluation of oracle bug-finding capability. We fix the default version of the fuzzer here (using the parameter settings as previously specified).

Consider first Figure 4 (a) Highway, where exact measuring lines by perfect oracles are availabe. These measuring lines attest to the strength of our metamorphic oracles in this domain: MMBug is close to PerfectBug, and MMSeedBugBasic is close to PerfectSeedBug. The average numbers of (seed-)bugs identified at the end of testing are 553.2 for MMBug, 588.7 for PerfectBug, 71.2 for MMSeedBugBasic, and 79.8 for PerfectSeedBug. Among the seed-bug oracles, FailureSeedBug reports many false positives, RuleSeedBug lags behind MMSeedBugBasic (40.3 at the end), and MMSeedBugExt finds a large number of additional bugs.

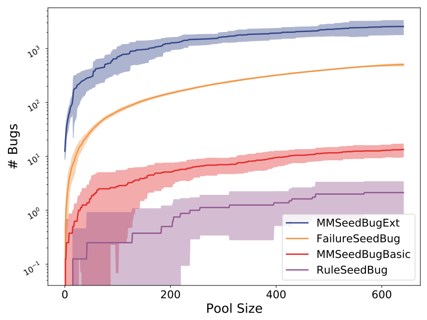
The gap between the seed-bug vs. bug oracles is large here because the latter identify many bugs among those pool states solved by the policy under the one seed chosen by the fuzzer (these pool states are ignored by the seed-bug



(a) Highway



(b) LunarLander



(c) BipedalWalker

Figure 4: Evaluation of oracles: number of (unique) bugs as function of pool size as the testing process progresses. MMSeed-Bug2Bug and MMBug not included in BipedalWalker as this is deterministic so these oracles coincide with MMSeedBugBasic.

SETTING / DOMAIN	`DIV_THRESH>0`								`DIV_THRESH=0`							
	`POL_PROB=0.2`				`POL_PROB=0`				`POL_PROB=0.2`				`POL_PROB=0`			
	# Bugs	**Avg. Dist.** (L_2)			**# Bugs**	**Avg. Dist.** (L_2)			**# Bugs**	**Avg. Dist.** (L_2)			**# Bugs**	**Avg. Dist.** (L_2)		
		Min	Max	Avg		Min	Max	Avg		Min	Max	Avg		Min	Max	Avg
Highway	71.2	3.6	62.8	12.1	66.5	3.6	62.3	13.8	96.2	0.9	51.9	12.8	98.0	0.6	49.8	11.3
LunarLander	59.1	0.6	5.0	1.8	22.5	0.6	2.98	1.4	336.2	0>	2.9	1.1	198.7	0>	2.9	1.1
BipedalWalker	13.3	2.0	5.9	3.7	13.8	2.1	6.1	3.7	12.8	1.0	4.0	2.7	19.1	1.0	4.2	2.6

Table 1: Evaluation of fuzzer settings: number and diversity of bugs at end of testing process, using MMSeedBugBasic.

oracles). Accordingly, while MMSeedBug2Bug is close to MMSeedBugBasic showing that most seed-bugs we identify are bugs, MMSeedBug2Bug lags far behind MMBug.

Consider now LunarLander and BipedalWalker in Figure 4 (b) and (c). MMSeedBugBasic vastly outperforms RuleSeedBug. FailureSeedBug is far above that, but at least in LunarLander this is again due to false positives: using search on the test-pool states where the policy fails, we found that at least 50% of these failures are unavoidable. MMSeedBugExt finds many additional bugs as before.

In LunarLander, MMBug is only slightly above MMSeed-BugBasic (in difference to Highway); MMSeedBug2Bug and MMSeedBugBasic are so close to each other that the two plots cannot be distinguished (98% of the seed-bugs reported by MMSeedBugBasic are bugs here). In Bipedal-Walker, the three plots necessarily coincide.

Overall, the results show that metamorphic oracles are superior to the rule-based and failure-based alternatives we evaluate; in the one domain where we are able to check, the oracles are close to perfect. Seed-bug detection is a practical proxy for bug detection in the sense that most seed-bugs detected by MMSeedBugBasic are bugs.

7.3 Results: Fuzzer Configurations

For our evaluation of fuzzer configurations – specifically the `POL_PROB` and `DIV_THRESH` algorithm parameters, which are most interesting as discussed – see Table 1.

Consider first the impact of `POL_PROB`, controlling whether or not the policy under test is used to (partially) inform the random walks in the fuzzer. This is intended to improve bug-finding capability in domains where purely random walks incur too many unavoidable failures. In our three case studies, this effect is indeed visible in LunarLander, where non-zero `POL_PROB` results in finding more than twice as many bugs for each setting of `DIV_THRESH`.

Non-zero `DIV_THRESH`, on the other hand, reduces the number of bugs found in LunarLander by up to an order of magnitude, with minor reductions in Highway and Bipedal-Walker. This is because more time is needed to fill the pool. The desired effect of increasing bug diversity is achieved though, with all min/max/average values being higher for non-zero vs. zero with the single exception of the average with `POL_PROB`=0.2 in Highway. An interesting synergy between non-zero `DIV_THRESH` and `POL_PROB` is manifested in LunarLander, where the larger number of bugs thanks to `POL_PROB` also results in higher diversity with (but not without) non-zero `DIV_THRESH`.

8 Conclusion

To test action policies for avoidable failures, oracles are required that can effectively identify sub-optimal behavior. We have shown that such oracles can be obtained from relaxations, by adapting ideas from metamorphic testing. Our experiments confirm the potential of this approach.

This works opens up an entire universe of exciting research on relaxation-based metamorphic oracles. Possibilities include the automated design of state relaxations and thus metamorphic oracles; intelligent methods to explore environment behaviors in a search for seed-bugs; fault localization trying to identify specific combinations of policy decisions leading to failures; as well as closing the loop with re-training by feeding bug states back into RL, until testing yields sufficient confidence in the policy.

9 Acknowledgments

This work was funded by DFG Grant 389792660 as part of TRR 248 (CPEC, https://perspicuous-computing.science).

References

Akazaki, T.; Liu, S.; Yamagata, Y.; Duan, Y.; and Hao, J. 2018. Falsification of Cyber-Physical Systems Using Deep Reinforcement Learning. In *FM*.

Bonet, B.; and Geffner, H. 2001. Planning as Heuristic Search. *AIJ*, 129: 5–33.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. OpenAI Gym. *CoRR*, abs/1606.01540.

Chen, T. Y.; Cheung, S. C.; and Yiu, S. 1998. Metamorphic Testing: A New Approach for Generating Next Test Cases. Technical Report HKUST–CS98–01, HKUST.

Corso, A.; Moss, R.; Lee, R.; and Kochenderfer, M. J. 2021. A Survey of Algorithms for Black-Box Safety Validation of Cyber-Physical Systems. *JAIR*, 72: 377–428.

Deng, Y.; Zheng, X.; Zhang, T.; Lou, G.; liu, H.; and Kim, M. 2020. RMT: Rule-based Metamorphic Testing for Autonomous Driving Models. *CoRR*, abs/2012.10672.

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-Black Planning: A New Systematic Approach to Partial Delete Relaxation. *AIJ*, 221: 73–114.

Dreossi, T.; Dang, T.; Donzé, A.; Kapinski, J.; Jin, X.; and Deshmukh, J. V. 2015. Efficient Guiding Strategies for Testing of Temporal Properties of Hybrid Systems. In *NFM*.

Edelkamp, S. 2001. Planning with Pattern Databases. In *ECP*, 13–24. AAAI.

Ernst, G.; Sedwards, S.; Zhang, Z.; and Hasuo, I. 2019. Fast Falsification of Hybrid Systems Using Probabilistically Adaptive Input. In *QEST*.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In *ICAPS*.

Gentilini, R.; Piazza, C.; and Policriti, A. 2003. From Bisimulation to Simulation: Coarsest Partition Problems. *J. Autom. Reason.*, 31: 73–103.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *ICAPS*.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *JACM*, 61: 16:1–16:63.

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems. In *ICAPS*, 422–430. AAAI.

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *AAAI*, 8064–8073. AAAI.

Koren, M.; Alsaif, S.; Lee, R.; and Kochenderfer, M. J. 2018. Adaptive Stress Testing for Autonomous Vehicles. In *IV*.

Lee, R.; Mengshoel, O. J.; Saksena, A.; Gardner, R. W.; Genin, D.; Silbermann, J.; Owen, M. P.; and Kochenderfer, M. J. 2020. Adaptive Stress Testing: Finding Likely Failure Events with Reinforcement Learning. *JAIR*, 69: 1165–1201.

Miller, B. P.; Fredriksen, L.; and So, B. 1990. An Empirical Study of the Reliability of UNIX Utilities. *CACM*, 33: 32–44.

Milner, R. 1971. An Algebraic Definition of Simulation Between Programs. In *IJCAI*, 481–489.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M. A.; Fidjeland, A.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533.

Odena, A.; Olsson, C.; Andersen, D.; and Goodfellow, I. J. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *ICML*.

Raffin, A.; Hill, A.; Ernestus, M.; Gleave, A.; Kanervisto, A.; and Dormann, N. 2019. Stable Baselines3. https://github.com/DLR-RM/stable-baselines3.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms. *CoRR*, abs/1707.06347.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T. P.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529: 484–489.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go Through Self-Play. *Science*, 362: 1140–1144.

Steinmetz, M.; Gros, T.; Heim, P.; Höller, D.; and Hoffmann, J. 2021. Debugging a Policy: A Framework for Automatic Action Policy Testing. In *ICAPS Workshop PRL*.

Tian, Y.; Pei, K.; Jana, S.; and Ray, B. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *Intl Conf Software Enginering*.

Torralba, Á.; and Hoffmann, J. 2015. Simulation-Based Admissible Dominance Pruning. In *IJCAI*.

Toyer, S.; Thiébaux, S.; Trevizan, F. W.; and Xie, L. 2020. ASNets: Deep Learning for Generalised Planning. *JAIR*, 68: 1–68.

Zhang, M.; Zhang, Y.; Zhang, L.; Liu, C.; and Khurshid, S. 2018. DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *Intl Conf Automated Software Engineering*.