Policy Safety Testing in Non-Deterministic Planning: Fuzzing, Test Oracles, Fault Analysis

Chaahat Jain[a,*], Daniel Sherbakov[a], Marcel Vinzent[a], Marcel Steinmetz[c], Jesse Davis[b] and Jörg Hoffmann[a,d]

[a]Saarland University, Saarland Informatics Campus, Germany
[b]Department of Computer Science, KU Leuven, Leuven, Belgium
[c]LAAS-CNRS, Toulouse, France
[d]German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany

Abstract. Recent work has introduced methodology for testing learned action policies in AI Planning, aiming to effectively identify bug states where policy behavior is sub-optimal. While this work focused on cost-optimality in classical planning, here we apply the core ideas to safety testing in planning with initial-state and action-outcome non-determinism. We cover the entire testing pipeline, introducing fuzzing algorithms to find unsafe policy runs, as well as test oracles to identify bugs where such unsafe behavior could be avoided. Going beyond the previous framework, we introduce a final step to the pipeline, identifying *faults* which we define to be specific policy decisions – state/action pairs – transitioning from a safe state (where a safe policy exists) to an unsafe state (where no such policy exists). We adapt a range of known algorithms for these purposes, including also approximate ones bounding the number of times we are allowed to diverge from the learned policy. We run comprehensive experiments evaluating each part of our pipeline. Key takeaways are that safety testing can be quite cheap, in contrast to cost-optimality testing; and that variants of Tarjan's algorithm tend to be highly effective for this purpose.

1 Introduction

Learned action policies, in particular neural ones, are gaining traction in AI [e.g., 25, 28, 29], including in AI Planning [20, 15, 14, 35, 22, 30, 31, 27]. However, such policies come without any built-in guarantees, so methods for quality assurance are important. One natural approach is policy testing to identify deficiencies in policy behavior.

Steinmetz et al. [32] introduced a framework for testing learned action policies π in AI Planning, defining a **bug** as a state s on which π behaves sub-optimally with respect to a **testing objective**. Their focus was on classical (deterministic) planning with cost-optimality as the testing objective, finding bugs s where either π does not find a plan though one exists, or π's plan has sub-optimal costs. Follow-up works introduced **fuzzing** methods (biased random walks) to generate candidate states s [11]; as well as **test oracles** to prove that s is a bug without having to run an optimal planner [10].

Here we apply the ideas of this framework to safety testing in non-deterministic planning, specifically planning with initial-state and action-outcome non-determinism. A given policy π is **safe** in a given state s if, for all possible runs of π from s (for all possible action outcomes), we never enter a state that satisfies a given **unsafety condition** ϕ_u. A state is **safe** if such a policy exists. Under our testing objective, a state s is a bug if π is unsafe on s although s is safe, i.e., if π's unsafe behavior on s could be avoided.[1]

We cover the entire testing pipeline. First, we introduce a form of fuzzing algorithms that find unsafe policy runs by biased sampling of action outcomes, leveraging a heuristic unsafety-distance function on the outcome states. Second, we devise test oracles which, given a state s on which π is unsafe, run searches trying to prove that s is safe (thereby proving that s is a bug). Third, we extend Steinmetz et al.'s framework with a final step in the testing pipeline. For any testing objective, if s is a bug, all we know is that somewhere below s the policy is sub-optimal (in our case: is avoidably unsafe). But what specific policy decisions are causing the sub-optimal behavior? Here we address this question in the safety-testing context. We define a **fault** to be a state-action pair $(s, \pi(s))$ where s is safe but there exists an outcome state s' that is unsafe. We show how to find such faults by running the search algorithms underlying our test oracles backwards on the states along a policy run, caching information to avoid duplicate work.

Proving that a state is safe – finding a safe alternate policy π' – differs from standard non-deterministic planning (e.g. [7, 26]) in that the only role of the goal is as a terminal state beyond which we do not need to search. Furthermore, as in general determining safety or unsafety exactly is anticipated to be costly or infeasible, we are interested in approximate variants of the problem as well. To this end, we adopt an idea recently proposed in a completely different context [9] (speeding up goal-conflict explanations in classical planning). We fix a **radius** r around the given learned policy π, permitting any

* Corresponding Author. Email: jain@cs.uni-saarland.de

[1] Earlier work on policy testing outside the planning context also addressed safety [e.g., 8, 1, 23, 12, 24], but disregarding avoidability, identifying "bug" with unsafe behavior.

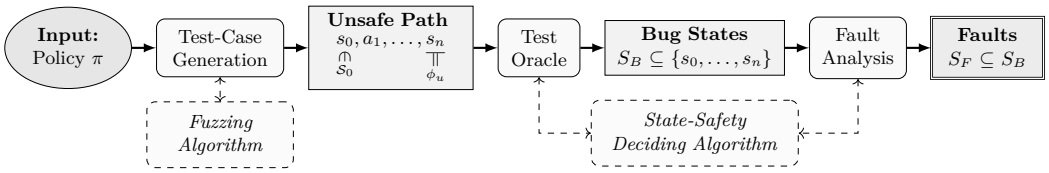

Figure 1. An overview of our testing pipeline.

run of π' to diverge from π at most r times (i.e., to take at most r different action decisions). We design exact algorithms solving this under-approximation, thereby obtaining sound but incomplete algorithms for deciding safety. These can be used as sufficient criteria in test oracles. Beyond that use, r-constrained safety is meaningful in its own right. If, e.g., π is unsafe in s but a 1-constrained safe policy π' exists, then we can "fix" π in s by changing a single decision – a strong statement about the bug s.

We formalize the r-constrained problem through a value function counting the maximal number of divergences from π along possible runs. We introduce a Bellman equation for this and show that its fixed points are optimal. Based on this, we introduce adapted variants of LRTDP [5], LAO* [16], and value iteration (VI) [4]. We furthermore show how to adapt Tarjan's algorithm; we refer to this adaptation as ***TarjanSafe***. Exact algorithms are part of this algorithm family, through setting $r = \infty$. Figure 1 illustrates our overall testing pipeline.

We run comprehensive experiments evaluating each part of our extended policy testing pipeline: fuzzing, test oracles, fault analysis. To this end, we adapt benchmarks previously considered for policy verification in planning contexts [36, 21] as well as control contexts [3, 2], and we add new benchmarks varying a transport problem with different forms of non-determinism. Key lessons learned are as follows. Our sampling bias in fuzzing finds many more unsafe policy runs than uniform sampling. Regarding oracles, ∞-*TarjanSafe* and 1-*TarjanSafe* can often find safe policies very fast and thereby form powerful quick test oracles. Regarding fault analysis, the main new challenge is that, to prove $(s, \pi(s))$ is a fault, we need to prove that for some outcome state s' no safe policy exists. ∞-*TarjanSafe* is still very competitive at doing so, but is outperformed by ∞-LRTDP on the most challenging instances. Approximation via 1-constrained unsafety is, as one would expect, less prone to long runtimes on unsafety proofs, making 1-constrained faults a reliable alternative.

2 Background

We consider fully-observable non-deterministic (FOND) planning tasks [7] in forms of non-deterministic transition systems $\Theta = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{S}_0, \mathcal{S}_* \rangle$, where $\mathcal{S}$ is a finite or infinite set of **states**; $\mathcal{A}$ is a set of **action labels**; $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightharpoonup (2^{\mathcal{S}} \setminus \{\emptyset\})$ is the non-deterministic **transition function**; $\emptyset \subset \mathcal{S}_0 \subseteq \mathcal{S}$ is a non-empty set of **initial states**, and $\mathcal{S}_* \subseteq \mathcal{S}$ is a set of **goal states**. Typically, the states are described only implicitly via Boolean, integer, or real-valued state variables, and actions are described via conditions and effects on those variables. We ignore this detail in the following, yet want to emphasize that Θ is usually too large to build and explore exhaustively. In our experiments, we consider different planning benchmarks provided in the JANI specification language [6], as considered in related works on policy verification [36, 21].

For each state s, $app(s) \subseteq \mathcal{A}$ denotes the set of actions **applicable** in s, i.e., the actions a where $\mathcal{T}(s, a)$ is defined. We assume without loss of generality that $app(s) \neq \emptyset$ holds for all states s. If $s' \in \mathcal{T}(s, a)$ is a possible successor of a in s, we also write $s \rightarrow_a s'$. A **path** is a finite or infinite alternating sequence of states and actions $\sigma = s_1, a_1, s_2, \ldots$ such that $a_i \in app(s_i)$ and $s_{i+1} \in \mathcal{T}(s_i, a_i)$ holds for all $i \geq 1$. The length of σ is denoted $|\sigma|$, where $|\sigma| = \infty$ if σ is infinite. If finite, $s_{|\sigma|}$ is the final or ending state of σ. σ is **maximal** if (1) $s_{|\sigma|} \in \mathcal{S}_*$ or σ is infinite, and (2) $s_i \notin \mathcal{S}_*$ for all $i < |\sigma|$. We refer to the set of all paths by *Paths* and the set of all maximal paths by *MaxPaths*. The (maximal) paths starting from a state s are referred to by *Paths*(s) (*MaxPaths*(s)). A **policy** is a function $\pi : \mathcal{S} \mapsto \mathcal{A}$ so that $\pi(s) \in app(s)$ for all states s. Π is the set of all policies. π induces the **policy graph** $\Theta^\pi = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}^\pi, \mathcal{S}_0, \mathcal{S}_* \rangle$ with the same components as Θ except for $\mathcal{T}^\pi$, which is given by $\mathcal{T}^\pi(s, a) := \mathcal{T}(s, a)$ if $a = \pi(s)$ and undefined otherwise. The set of paths $Paths^\pi$ (maximal paths $MaxPaths^\pi$) induced by π are the (maximal) paths of Θ^π.

An **unsafety condition** is a Boolean state predicate $\phi_u : \mathcal{S} \mapsto \mathbb{B}$. A state s is **unsafe** if $\phi_u(s)$ is true, written $s \models \phi_u$.

3 Our Policy Safety Testing Framework

We next introduce our policy safety testing framework, formalizing its basic definitions and discussing its intended use. As outlined in the introduction, we follow Steinmetz et al.'s [32] generic policy testing framework, instantiating it for safety in non-deterministic planning, and extending it with a final fault analysis step.

In a nutshell, Steinmetz et al.'s framework suggests to fix a **testing objective**, formalized by a value function V^π that maps states s to their value V^π under a given policy π. With V^* denoting the best value under any policy, a state s is then a **bug** if $|V^*(s) - V^\pi(s)| > 0$. For practical implementation of such policy testing, methods are required for 1. generating test cases s, and 2. proving a given s to be a bug. Steinmetz et al. and follow-up works [10, 11] focused on classical planning with cost-optimality as the testing objective. For 1., they design biased sampling (random walk) methods, geared at finding states s with bad policy performance. For 2., they design **test oracles**, sufficient criteria that can identify some but not all bugs without resorting to optimal planning. The latter is essential in their setting, as the testing objective coincides with the learning objective – if cost-optimal planning was feasible in the given domain, then there would be no need for learning a policy in the first place.

Here we tackle safety testing in non-deterministic planning. Our testing objective is:

Definition 1. *(π-safety objective) For a policy π and state s we define the value of π in s as*

$$V^{\pi}(s) = \begin{cases} 1 & \text{if } s \models \phi_u \\ 0 & \text{else if } s \in \mathcal{S}_* \\ \max_{s':s\to_{\pi(s)}s'} V^{\pi}(s') & \text{else} \end{cases}$$

The optimal value in s is given by $V^*(s) = \min_{\pi\in\Pi} V^{\pi}(s)$.

With this, our bug definition directly follows Steinmetz et al.'s framework:

Definition 2. *(π-safety and bugs) For a policy π and state s, we say that s is **safe** if $V^*(s) = 0$, and that π is **safe in** s if $V^{\pi}(s) = 0$. We say that s is a **bug** if $|V^*(s) - V^{\pi}(s)| > 0$.*

In other words, a bug here is a state s on which π is unsafe – an unsafe state can be reached under π – even though a safe policy for s exists. In this situation, π's unsafe behavior on s is **avoidable**. For example, in a driving domain, π may be unsafe on s because in one of the non-deterministic outcomes an object crashes into the car (unavoidable unsafety, not a bug), or π may be unsafe on s because it chooses to accelerate towards a wall (avoidable unsafety, bug).

Observe that, in Steinmetz et al.'s framework, and hence in the definitions we just gave, even if s is a bug state, *π's choice on s itself may be optimal*: all we know is that something goes wrong in the policy run beneath s. Considering again the driving example, if π chooses to accelerate towards a wall in state s, then all states preceding s in that policy run (e.g. when the car turned a corner 27 steps ago) are bugs as well. So how to define and find the "bad" policy decisions, that cause the undesired behavior? Steinmetz et al. call this **fault localization**, and leave it open for future work. Here we provide an instantiation for policy safety testing:

Definition 3. *(π-safety fault) For a policy π and state s, $(s, \pi(s))$ is a **fault** if s is safe but there exists a transition $s \to_{\pi(s)} s'$ where s' is unsafe.*

We deem a policy decision to be a fault if it takes the agent from the safe region, where unsafety can be avoided with certainty, into the unsafe region where that is not the case.

Some words are in order on the intended use of our framework, and some important computational implications. In contrast to classical planning and cost-optimality testing as in prior work, in our setting the testing objective does *not* coincide with the learning objective. To be safe, it suffices to avoid the unsafety condition; in contrast, policies will also be trained to reach the goal. Concretely, a canonical application setting for our form of policy testing is that where π was trained using RL to maximize expected discounted reward in an MDP where goal states give positive rewards while unsafe states give (high) negative rewards. This is a commonly used proxy objective for RL when goal reachability must be traded against safety [13, 37]. Our testing machinery then considers worst-case (non-deterministic) behavior in this MDP, and focuses entirely on safety, disregarding the goal except for safe termination of policy runs. From the conceptual side, this makes sense as a focused test on an important aspect of system performance; faults identified by our testing pipeline can inform human policy inspection, or further policy training. From the computational side, solving a state s "optimally" w.r.t. our testing objective means to prove whether or not s is safe. This is a hard problem in general, but may be more feasible in practice than maximizing rewards in the MDP where π was learned. Indeed, our experiments suggest that deciding safety can be quite easy in our setting.

In the remainder of the paper, we operationalize our policy safety testing framework as illustrated in Figure 1. Section 5 discusses fuzzing methods that generate test cases seeding the bug and fault analyses; Section 6 discusses the design of test oracles and fault detection methods based on state-safety deciding algorithms. Prior to these discussions, we introduce r-constrained safety as well as policy bugs/faults variants based on that (Section 4). These variants are of interest in their own right as argued; r-constrained safety furthermore serves for the design of approximate methods in Section 6.

4 r-Constrained Safety

Following prior work on approximate policy explanations [9], we consider approximate variants of Definitions 1 and 2 that base the sate-safety assessment on policies within proximity of the policy to be analyzed.

Definition 4 (r-Policy Space)**.** *Let π and π' be two policies, and s be a state. We consider the distance measure*

$$\delta_s(\pi, \pi') = \max_{\sigma=s_1,a_1,\dots\in MaxPaths^{\pi'}(s)} \sum_{i=1}^{|\sigma|} [\![\pi(s_i) \neq a_i]\!]$$

*where $[\![\cdot]\!]$ denotes the Iverson bracket. The policies within a **radius** $r \in \mathbb{Z}_0^+ \cup \{\infty\}$ from π in state s are denoted $\Pi^s_{\pi,r} = \{\pi' \in \Pi \mid \delta_s(\pi, \pi') \leq r\}$.*

In words, $\delta_s(\pi, \pi')$ measures the maximal number of policy-choice differences between π and π' along any maximal s-path induced by π'; and $\Pi^s_{\pi,r}$ contains all the policies differing from π no more than r times. We generalize Definitions 1 and 2 by adapting the optimal value function accordingly:

Definition 5 (r-Bug)**.** *For a policy π and a radius $r \in \mathbb{Z}_0^+ \cup \{\infty\}$, the r-constrained optimal safety value function is given by $V^*_{\pi,r}(s) = \min_{\pi'\in\Pi^s_{\pi,r}} V^{\pi'}(s)$. We say that s is r-**safe** on π if $V^*_{\pi,r}(s) = 0$, and that s is an r-**bug** if $|V^*_{\pi,r}(s) - V^{\pi}(s)| > 0$.*

Clearly, the notions of ∞-bugs and bugs are equivalent. For $r < \infty$, each r-bug is also a bug, but not necessarily vice versa. There can be bugs that are not r-bugs for any value of $r < \infty$. This is a direct consequence of the definition of δ_s, which entails $\delta_s(\pi, \pi') = \infty$ in case policy-choice differences are enclosed in some cycle. For $r < \infty$, r-bugs yield a sufficient bug-condition test. At the same time, verifying r-safety can be potentially easier by considering alternative options only in the very confined perimeter around the given policy as controlled by the radius parameter r. Besides being potentially computationally easier, Definition 5 also provides a novel way to classify bugs. As resolving r-bugs could be considered less expensive than r'-bugs for $r' > r$, this information might help domain experts in assessing the severity of different bugs.

On top of Definition 4, we also considered two alternative distance (and hence r-bug) definitions. First, in place of maximizing over paths, one could consider summation. This however introduces a dependency to the non-determinism present

Algorithm 1: Fuzzing algorithm

Data: Policy π, lookahead limit $D_{\text{limit}} \in \mathbb{Z}^+ \cup \{\infty\}$, $h_u : \mathcal{S} \mapsto \mathbb{Z}_0^+$ distance-to-unsafety estimation
Result: Path $s_1, \pi(s_1), s_2, \pi(s_2), \ldots, s_n$ such that $s_n \models \phi_u$ or "failed"

$s_1 \leftarrow$ sample from $\mathcal{S}_0$ uniformly at random;
$\sigma \leftarrow s_1$;
while $s_{|\sigma|} \not\models \phi_u$ **do**
  $\sigma' \leftarrow$ *FindContinuation*$(s_{|\sigma|})$;
  if σ' *is "failed"* **then**
    return *"failed"*
  $\sigma \leftarrow$ concatenate σ and σ';
return σ
Function *FindContinuation(s)*:
  for $d = 1, \ldots, D_{limit}$ **do**
    $S_d \leftarrow$ states d steps away from s in Θ^π;
    if $S_d = \emptyset$ **then**
      return *"failed"* `// exhausted successors`
    if $\exists s' \in S_d$*:* $s' \models \phi_u$ **then**
      return *policy path from s to s'*
    if $\operatorname{argmin}_{s' \in S_d} h_u(s')$ *is unique* **then**
      break;
  $s' \leftarrow \operatorname{argmin}_{s' \in S_d} h_u(s')$ or sample from $\bigcup_d S_d$;
  return *policy path from s to s'*

in a planning task, harming comparability and interpretability of the results, while also suffering from the same drawbacks as maximization. Another alternative is to count differences in the policy graph at global instead of path-individual level. In preliminary experiments, this turned out to be computationally very expensive. So, we focus on the distance measure and r-bug conditions as per Definitions 4 and 5.

5 Fuzzing: Test-Case Generation through Lookahead Search and Sampling

In order to identify policy bugs effectively, it is crucial to have test-generation methods that can quickly find policy runs leading to unsafe states. Given a policy that is believed to have reasonable quality, i.e., one being worthwhile to analyze in the first place, it is unlikely to find such runs by sampling policy executions uniformly at random. Instead, we follow the heuristic approach sketched in Algorithm 1, which leverages functions h_u estimating the distance from states towards the satisfaction of the unsafety condition, to guide the policy run generation. We discuss our choice of h_u below.

Algorithm 1 iteratively composes a policy execution path, starting from an initial state that is chosen uniformly at random. The path is incrementally extended through calls to *FindContinuation* until either an unsafe path has been constructed or an extension of the current path to an unsafe path is no longer possible. *FindContinuation* performs policy simulations and uses h_u to resolve successor choices left under the non-deterministic transition function. To escape plateaus of successor states indistinguishable by the provided distance estimates, *FindContinuation* resembles a variant of the hill climbing algorithm [17], running a breadth-first lookahead search in the policy graph Θ^π until (a) a unique descendant with minimal h_u value is found, or (b) an unsafe state is found, or (c) the policy sub-graph rooted at the considered state has been exhausted. A lookahead depth bound optionally allows to cap the computational overhead. In case of (c), the current policy path cannot be extended to an unsafe path, and *FindContinuation* as well as the main procedure return a failure code. In case of (b), *FindContinuation* returns the corresponding path suffix and the main procedure terminates. In case of (a), or if the depth limit was reached, *FindContinuation* extends the policy path to some visited state. We experimented with two strategies to select this state: **greedy**, we choose a state $s' \in S_d$ from the last state layer that is deemed closest to satisfying the unsafety condition as per h_u; or by **weighted distribution sampling**, where we sample from all seen states $s' \in \bigcup_d S_d$ according to the distance-weighted probability $p(s') = e^{-h_u(s')} / \sum_{s'' \in \bigcup_d S_d} e^{-h_u(s'')}$.

Suitable distance functions h_u can for example be found in the vast research on planning heuristics. However, due to their lack of support of Jani planning task models, in our experiments, we instead fall back on a simple function inspired from the Manhattan distance. In our case, states are vectors of real-valued variable-value assignments, and the unsafety condition ϕ_u is a Boolean combination of linear constraints. We estimate the distance $h(s, \phi_u)$ from a state s to the satisfaction of ϕ_u through $h(s, \phi) = 0$ if $s \models \phi$; $h(s, \phi) = |\sum_i w_i s(x_i) - b|$ for the linear constraint $\phi : \sum_i w_i x_i \leq b$; and decompose conjunctions and disjunctions by using summation and minimization over the decompositions respectively.

6 State-Safety Deciding Algorithms for Test Oracles and Fault Analysis

After having identified an unsafe policy path σ, we proceed in the testing pipeline (cf. Fig. 1) with the analysis of the states visited on σ to determine which ones are indeed policy bugs and which ones are faults. This analysis relies on methods that decide whether a given state is safe, i.e., whether a different policy can guarantee to avoid the unsafety condition when run from that state. Such methods are discussed next. Note that, due to non-determinism, σ can contain multiple faults. In order to identify all faults efficiently, we share state-safety information between the fault checks, avoiding redundant computations, and process σ in reverse, which tends to make use of this shared information effectively.

To decide state safety efficiently, we developed two orthogonal approaches: via well-known MDP algorithms, respectively a search algorithm geared for this purpose. Our methods decide r-safety in general, which as previously pointed out, subsumes safety for $r = \infty$.

6.1 MDP Algorithms

Non-deterministic transition systems can be seen as MDPs with unspecified transition probabilities. We next show how to use standard MDP algorithms for computing a proxy to the optimal safety value function, based on an MDP objective that characterizes r-safety and is amenable to optimization via common dynamic programming approaches.

The core idea is combining state and action costs, where action costs reflect deviations from the policy under testing

and a termination cost penalizes unsafe states. We are interested in the MDP policy solution that minimizes the worst-case summed up cost over all runs of that policy. Formally, this optimal solution is characterized by the piecewise smallest function $J^*_{\pi,r}$ that satisfies $J^*_{\pi,r}(s) = \mathcal{B}_{\pi,r}J^*_{\pi,r}(s)$, for all states s, under the Bellman operator $\mathcal{B}_{\pi,r}J(s) :=$

$$\begin{cases} \mathsf{U}_r & \text{if } s \models \phi_u \\ 0 & \text{else if } s \in \mathcal{S}_* \\ \min_{a \in app(s)} \left(\mathsf{C}_{\pi,r}(s,a) + \max_{s' \in \mathcal{T}(s,a)} J(s') \right) & \text{else} \end{cases}$$

where $J : \mathcal{S} \mapsto \mathbb{R}^+_0$ is any function, $\mathsf{U}_r = r + 1$ if $r < \infty$ and $\mathsf{U}_r = 1$ else, which gives the cost of reaching an unsafe state; and $\mathsf{C}_{\pi,r}(s,a) = 1$ if $r < \infty$ and $\pi(s) \neq a$, $\mathsf{C}_{\pi,r}(s,a) = 0$ otherwise, which represents the policy deviation cost. Goal states and unsafe states are considered to be terminating. It is straightforward to show that r-safety on the policy π under testing is characterized exactly by the $J^*_{\pi,r}$ values:

Theorem 1. *For all policies π, states s, and radii $r < \infty$, s is r-safe on π iff $J^*_{\pi,r}(s) \leq r$. s is ∞-safe iff $J^*_{\pi,\infty}(s) = 0$.*

$J^*_{\pi,r}$ can be computed via value iteration and MDP heuristic search algorithms (e.g., LAO* [16] or LRTDP [5]). In a nutshell, those approaches start from some cost-value initialization $J^{(0)}$ and repeatedly apply Bellman updates $J^{(i)} := \mathcal{B}_{\pi,r}J^{(i-1)}$ until a fixed point $J^{(k)} = J^{(k-1)}$ is reached.

The standard correctness argument establishing $J^{(k)} = J^*_{\pi,r}$ upon termination requires (1) the actual existence of a fixed point, and (2) that this fixed point coincides with $J^*_{\pi,r}$. However, *it is not immediately evident why $\mathcal{B}_{\pi,r}$ from above would satisfy either of the properties.* Namely, while the definition of $\mathcal{B}_{\pi,r}$ closely resembles the standard cost-minimization MDP objective, there is the important difference that we do not enforce any requirement on reaching goal states eventually (cycling forever while avoiding unsafety is a valid strategy). Additionally, as opposed to general reward-maximization MDP objectives, we consider the optimization over an infinite horizon without discount factor.

That there actually exists any fixed point, (1), follows from the Knaster-Tarski theorem [34] and the fact that $J^*_{\pi,r}$ is bounded from above by U_r and that the cost-value functions $\hat{J}^{(0)}, \hat{J}^{(1)}, \ldots$ increase monotonically, if one starts from $\hat{J}^{(0)}(s) := 0$, where $\hat{J}^{(i)} := \mathcal{B}_{\pi,r}\hat{J}^{(i-1)}$ for $i \geq 1$. (2) is actually not satisfied in general. In our case, there can be multiple solutions to the Bellman equation $J = \mathcal{B}_{\pi,r}J$ due to 0-cost cycles. However, by definition, $J^*_{\pi,r}$ is the least of those solutions. So, given that $\hat{J}^{(0)} \leq J^*_{\pi,r}$ and that $J \leq J^*_{\pi,r}$ implies $\mathcal{B}_{\pi,r}J \leq \mathcal{B}_{\pi,r}J^*_{\pi,r} = J^*_{\pi,r}$, the sequence of cost-value functions starting from 0 must converge to $J^*_{\pi,r}$ necessarily:

Theorem 2. *Let $\hat{J}^{(0)}(s) := 0$ and $\hat{J}^{(i)}(s) := \mathcal{B}_{\pi,r}\hat{J}^{(i-1)}(s)$ for $i \geq 1$ and all states s. Then $\lim_{i \mapsto \infty} \hat{J}^{(i)}(s) = J^*_{\pi,r}(s)$.*

Given this property, existing MDP algorithms like value iteration and heuristic search can be applied as is, as long as the cost-value function is initialized to 0.

6.2 r-Safety Search

On top of the general purpose MDP algorithms just presented, we further designed a method that is specifically tailored to proving r-safety quickly. Algorithm 2 shows the pseudocode, referred to by *TarjanSafe* in the following. The core procedure *FindSafePolicy*(s, b_s) runs a backtracking search to check whether a safe policy for s exists within the "policy-divergence budget" b_s that is remaining for s according to the recursion path started from the initial call arguments $\langle s_0, r \rangle$. The method greedily tries to construct a safe policy for s by testing in turn each possible action that is applicable in s and whose selection remains within the remaining budget. For each such action a, *FindSafePolicy* determines safety by calling itself on all the non-deterministic successor states s' of applying a in s with the remaining budget b' updated from b_s accordingly. When a safe action is found within the budget, *FindSafePolicy* terminates. Otherwise, *FindSafePolicy* continues until having exhausted all possibilities.

Algorithm 2: *TarjanSafe* algorithm deciding r-safety

```
Data: Policy π, safety radius r ∈ ℤ⁺₀ ∪ {∞}, state s₀
Result: True if s₀ is r-safe on π, and false otherwise
1  yes[s] ← ∞ for all states s;    // cache pos. results
2  no[s] ← −∞ for all states s;    // cache neg. results
3  b_{s₀} ← r;                     // initial "budget"
4  return FindSafePolicy(s₀, b_{s₀})
5  Function FindSafePolicy(s, b_s):
6      if s ⊨ φ_u or no[s] ≥ b_s then return false ;
7      if s ∈ S_* or yes[s] ≤ b_s then return true ;
8      allUnsafe ← true;
9      foreach a ∈ app(s) and while allUnsafe do
10         b' ← b_s − [[π(s) ≠ a]];
11         if b' < 0 then continue;
12         isSafe ← true;
13         foreach s' ∈ T(s, a) and while isSafe do
14             if s' with budget b' has not been expanded or
                 ⟨s', b'⟩ is not in current call's SCC then
15                 isSafe ← FindSafePolicy(s', b');
16         allUnsafe ← not isSafe;
17     if allUnsafe then no[s] = b_s ;
18     else
19         if ⟨s, b_s⟩ is the root of its SCC then
20             yes[s] = min{yes[s], b_s};
21     return not allUnsafe
```

If no safe action was found, the state is provably not b_s-safe on π. On the other hand, however, if an action that is potentially safe is found, the actual b_s-safety status might depend on the result of some parent call *FindSafePolicy*$(s', b_{s'})$, whose safety confirmation is currently still in progress. This situation arises exactly when the chosen action at s introduces a cycle to some state-budget pair $\langle s', b' \rangle$ still under expansion. We identify such cycles by incorporating Tarjan's algorithm for finding maximal SCCs [33] (hence the name *TarjanSafe*). If *FindSafePolicy*(s, b_s) returns true while the state-budget pair $\langle s, b_s \rangle$ marks the entry point into an SCC, there are no dependencies to the pending parent *FindSafePolicy* calls. Hence, at this point (and only at this point), we know that s is b_s-safe on π. We want to remark in particular that for the initial call on $\langle s_0, r \rangle$, *FindSafePolicy* always determines s_0's r-safety status, as this call never has dependencies.

To speed up the process and to avoid redundant computations, we cache intermediate b-safety respectively non-b-safety results in forms of the minimal budget $\texttt{yes}[s]$ for each state

s for which a `yes`$[s]$-safe policy for s has been found, respectively the maximal budget `no`$[s]$ for which s was proven to be not `no`$[s]$-safe. Both caches are updated accordingly when returning from *FindSafePolicy* using the computed safety status information as discussed in the previous paragraph.

In summary, our method guarantees:

Theorem 3. *For every policy π, state s_0, and radius $r \in \mathbb{Z}_0^+ \cup \{\infty\}$, Algorithm 2 returns true iff s_0 is r-safe on π.*

7 Experiments

Our implementation is in C++ and is available online[2]. We evaluate each component of the testing pipeline (cf. Fig. 1) in isolation, proceeding in the order: 1. fuzzing (test-case generation); 2. test-oracle performance of the safety-deciding algorithms, considering individual states, and 3. performance of fault analysis as a whole, processing an entire unsafe policy path. For 2. and 3., we use the test cases generated by our best fuzzing method. We next describe our benchmarks, before diving into the empirical results. All experiments were run on machines with Intel Xeon E5-2660 CPUs.

7.1 Benchmarks

A testing-benchmark instance is a pair of non-deterministic planning task described in the JANI language [6] and policy solving that problem. Our benchmark set is composed of three parts: (B1) benchmarks (beluga, and two variants of blocksworld and n-puzzle) from prior work on action-policy safety assessment [36, 21]; (B2) we created new JANI benchmarks, modeling the control problems *Bouncing ball, Follow Car, Cartpole* and *Inverted Pendulum* often considered in reinforcement learning [3, 2]; and (B3) two completely new benchmarks: *one-way line* and *two-way line*. In the latter benchmarks, a truck moves along a discrete line in one respectively both directions. The truck can accelerate and decelerate by one speed unit at a time, and pick up and drop packages if its velocity is 0. Non-determinism might cause the truck to drop packages while moving. Acceleration and deceleration might fail. We additionally consider variants with an additional parking action and one disabling non-determinism. The safety constraint requires not driving past either ends of the line. The planning tasks from (B2) are deterministic and contain bounded real-valued state variables; all other benchmarks are non-deterministic and use bounded integer state variables.

For each JANI benchmark instance, we train multiple feed-forward neural-network policies via Q-learning, similar to prior work [36]. We use ReLU activation and experimented with hidden layer sizes $\{16, 32, 64, 128, 256\}$, choosing for each JANI benchmark instance the best-performing policy which was not safe. For the benchmarks from (B1), we did not train new policies, but used those provided by [21].

7.2 Fuzzing

We ran a total of 5 configurations of our fuzzing algorithm (Algorithm 1), considering lookahead depths of $D_{\text{limit}} \in \{1, \infty\}$ and three different successor selection strategies: **uniform**, a baseline which samples successors uniformly at random; **greedy** and weighted distribution **sampling**, i.e., the two strategies introduced in Section 5. For uniform sampling, we did not run $D_{\text{limit}} = \infty$, which does not make sense. For each testing-benchmark instance, each configuration is executed 1000 times. Fig. 2 reports the results.

The control benchmarks from (B2) are deterministic, in which case, all the fuzzing algorithm configurations behave identically. In blocksworld, no unsafe policy path was found by any configuration. In all other domains, the unsafety distance estimates have a clear positive effect on test-case generation. That advantage is especially evident for the depth-unlimited variants, which however also come with the by far most significant overhead. Sampling versus greedily choosing successor states makes a difference primarily if the lookahead is bounded. This makes sense, intuitively, as distance estimates become more accurate the deeper the lookahead, reducing the benefits of the additional randomness introduced by sampling as the depth-bound is increased. All remaining experiments are based on the test cases generated by ∞-greedy.

7.3 Test Oracles

We evaluated the performance of 4 different safety-deciding algorithms – value iteration, LRTDP [5], LAO* [16], and *TarjanSafe* – for deciding (∞-)safety. We additionally consider each of the algorithms to compute the r-safety approximation, for $r \in \{1, 2\}$. We compare their performance to prove whether an individual state is a bug. To this end, we collect for each benchmark instance all states from the generated test cases. To balance state numbers between instances, we pick for each benchmark instance the whole collection, if that contains at most 1000 states, and pick from the collection 1000 states at random otherwise. We run every algorithm configuration on every state of the post-processed collection of all the benchmark instances, enforcing memory and time limits for 12 GB and 30 minutes per run. Value iteration almost always exceeded the memory budget during its state-space construction process, and is omitted in the following.

Consider first the coverage plots in Fig. 3. Regardless of the algorithm, bug confirmation tends to be cheap. Almost all test states were processed within split seconds. Comparing the algorithms, ∞-*TarjanSafe* turned out to be most effective for both proving and disproving whether a state is a bug. For the former cases, we observed that the search often was lucky in quickly finding alternative action choices that, e.g., via forming cycles, guarantee maintaining safety. For the latter cases, ∞-*TarjanSafe*' caching feature turned out highly effective. ∞-LRTDP offers a similar performance profile, but has a small but consistent runtime disadvantage. Not visible in the coverage plots, ∞-LRTDP however turned out slightly more effective than ∞-*TarjanSafe* in the very hard cases proving that a state is indeed not safe, i.e., is not a bug. Maybe most surprisingly, the r-safety variants for finite r seem to not offer any advantage in terms of efficiency, yet overall, identify much fewer bug states due to their under-approximation nature.

The runtime plots in Fig. 3 provide additional information. Consider the one for the bug states. ∞-*TarjanSafe* proves most bug-states safe within split seconds by quickly finding

[2] link omitted to preserve anonymity

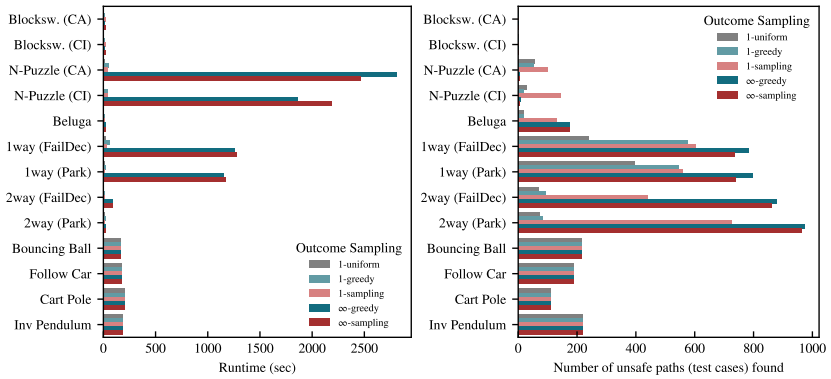

Figure 2. Per-domain comparison of different fuzzing algorithm configurations. The left plot shows the total time to run the algorithms for 1000 times. The right plot shows the number of unsafe policy paths found in these runs.

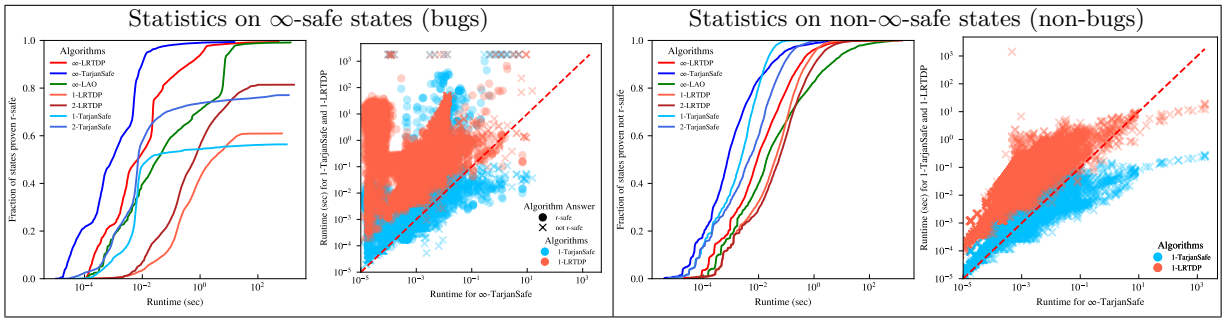

Figure 3. Test-oracle performance statistics for the bugs in the test-state collection respectively the non-bugs. Left plot in each category shows coverage in forms of the fraction of (non-)bug states proven (not-)r-safe over time. Right plot compares runtime (s) per test state between ∞-*TarjanSafe* (x-axis) and 1-LRTDP and 1-*TarjanSafe* (y-axis).

safety-guaranteeing cycles. However, to prove 1-safety, policy deviations cannot be included in cycles, which makes bug confirmation significantly harder despite allowing even just one policy deviation per policy execution run. Moreover, as indicated by the X entries, 1-safety is insufficient for considerable fraction of the bug states. On the other hand, proving that states are not 1-safe is almost always significantly cheaper than proving non-∞-safety. This is expected, given that disproving ∞-safety requires the exploration of all the policy alternatives whereas the option space for 1-safety is much more constrained. One needs to keep in mind, however, that non-1-safety is weaker than non-∞-safety in that non-1-safety does not necessarily imply that a state is not a bug.

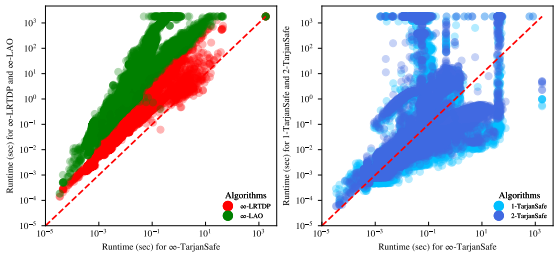

Figure 4. Fault analysis: runtime (s) of processing an entire test case of ∞-*TarjanSafe* (x-axis) vs. ∞-LRTDP/LAO* (left plot y-axis) and r-*TarjanSafe* (right plot y-axis).

7.4 Fault Analysis

Lastly, we evaluate the performance of the previously considered safety-deciding algorithms to identify all faults on an entire unsafe-path test case. The algorithms maintain a single cache (value function), which is continuously updated while processing each state on the unsafe path. In contrast to the test-oracle performance evaluation, we also evaluate how effectively each algorithm uses this caching functionality across multiple safety-decision calls.

Fig. 4 provides a runtime comparison between the different algorithms. Among the ∞-safety deciding algorithms, *TarjanSafe* carries over its test-oracle performance, having a consistent and considerable advantage over LRTDP (and a significant one over LAO*). Different from the previous results, fault-analysis performance of the approximative r-*TarjanSafe* variants turned out highly complementary to ∞-*TarjanSafe*. In the vertical lines, ∞-*TarjanSafe* shows its strength in proving ∞-safety quickly. In the cases below the diagonal, the approximative r-*TarjanSafe* exploit the stronger requirements of proving states not-r-safe. ∞-*TarjanSafe* identified at least one fault in 86% of the test cases. In almost all test cases, there was just a single fault, but there are cases where more than one fault was found. The latter can be observed to a much higher degree for the approximative variants, indicating some variability of the difficulty of resolving different bugs on the same unsafe path.

8 Conclusion

Testing is a natural method for quality assurance of learned action policies π. Here we transfer ideas from prior work to safety testing in non-deterministic planning, extending the testing pipeline with an additional fault-detection step, and covering approximate analyses via constraining alternate policies to a radius around π. Adapting a broad range of algorithms for these purposes, our empirical results indicate that this form of policy testing can be quite feasible. As one key outcome of this research and our implemented machinery, we are now able to identify specific faults in a given learned policy. This opens the possibility, for future work, to improve the policy leveraging this fault information. There are manifold possibilities for this, ranging from shielding over continued RL to neurosymbolic methods facilitating a guarantee to avoid known faults.

References

[1] T. Akazaki, S. Liu, Y. Yamagata, Y. Duan, and J. Hao. Falsification of cyber-physical systems using deep reinforcement learning. In *22nd International Symposium on Formal Methods (FM'18)*, pages 456–465, 2018.

[2] E. Bacci and D. Parker. Verified probabilistic policies for deep reinforcement learning. In *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *LNCS*, pages 193–212. Springer, 2022.

[3] E. Bacci, M. Giacobbe, and D. Parker. Verifying reinforcement learning up to infinity. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 2154–2160, 2021.

[4] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[5] B. Bonet and H. Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *ICAPS*, pages 12–21, 2003.

[6] C. E. Budde, C. Dehnert, E. M. Hahn, A. Hartmanns, S. Junges, and A. Turrini. JANI: quantitative model and tool interaction. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*, pages 151–168, 2017.

[7] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. 147(1–2):35–84, 2003.

[8] T. Dreossi, T. Dang, A. Donzé, J. Kapinski, X. Jin, and J. V. Deshmukh. Efficient guiding strategies for testing of temporal properties of hybrid systems. In *7th International Symposium NASA Formal Methods (NFM'15)*, pages 127–142, 2015.

[9] R. Eifler, D. Fiser, A. Siji, and J. Hoffmann. Iterative oversubscription planning with goal-conflict explanations: Scaling up through policy-guidance approximation. In *Proc. ECAI*, pages 4092–4099, 2024.

[10] J. Eisenhut, A. Torralba, M. Christakis, and J. Hoffmann. Automatic metamorphic test oracles for action-policy testing. In *ICAPS*, pages 109–117, 2023.

[11] J. Eisenhut, X. Schuler, D. Fišer, D. Höller, M. Christakis, and J. Hoffmann. New fuzzing biases for action policy testing. In *ICAPS*, 2024.

[12] G. Ernst, S. Sedwards, Z. Zhang, and I. Hasuo. Fast falsification of hybrid systems using probabilistically adaptive input. In *16th International Conference on Quantitative Evaluation of Systems (QEST'19)*, pages 165–181, 2019.

[13] J. García and F. Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16:1437–1480, 2015.

[14] S. Garg, A. Bajpai, et al. Size independent neural transfer for RDDL planning. In *ICAPS*, pages 631–636, 2019.

[15] E. Groshev, M. Goldstein, A. Tamar, S. Srivastava, and P. Abbeel. Learning generalized reactive policies using deep neural networks. In ICAPS-18 [18], pages 408–416.

[16] E. A. Hansen and S. Zilberstein. LAO*: a heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.

[17] J. Hoffmann and B. Nebel. The ff planning system: Fast plan generation through heuristic search. pages 253–302, 2001.

[18] ICAPS-18. *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS'18)*, 2018.

[19] ICAPS-22. *Proceedings of the 32th International Conference on Automated Planning and Scheduling (ICAPS'22)*, 2022.

[20] M. Issakkimuthu, A. Fern, and P. Tadepalli. Training deep reactive policies for probabilistic planning problems. In ICAPS-18 [18], pages 422–430.

[21] C. Jain, L. Cascioli, L. Devos, M. Vinzent, M. Steinmetz, J. Davis, and J. Hoffmann. Safety verification of tree-ensemble policies via predicate abstraction. In *Proceedings of the 27th European Conference on Artificial Intelligence (ECAI'24)*, 2024.

[22] R. Karia and S. Srivastava. Learning generalized relational heuristic networks for model-agnostic planning. In *AAAI*, pages 8064–8073, 2021.

[23] M. Koren, S. Alsaif, R. Lee, and M. J. Kochenderfer. Adaptive stress testing for autonomous vehicles. In *IEEE Intelligent Vehicles Symposium (IV'18)*, pages 1–7. IEEE, 2018.

[24] R. Lee, O. J. Mengshoel, A. Saksena, R. W. Gardner, D. Genin, J. Silbermann, M. P. Owen, and M. J. Kochenderfer. Adaptive stress testing: Finding likely failure events with reinforcement learning. *Journal of Artificial Intelligence Research*, 69:1165–1201, 2020.

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

[26] C. J. Muise, S. A. McIlraith, and J. C. Beck. Improved nondeterministic planning by exploiting state relevance. In *Proc. ICAPS*, 2012.

[27] N. Rossetti, M. Tummolo, A. E. Gerevini, L. Putelli, I. Serina, M. Chiari, and M. Olivato. Learning general policies for planning through gpt models. *Proceedings of the International Conference on Automated Planning and Scheduling*, 34 (1):500–508, May 2024.

[28] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[29] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[30] S. Ståhlberg, B. Bonet, and H. Geffner. Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits. In ICAPS-22 [19], pages 629–637.

[31] S. Ståhlberg, B. Bonet, and H. Geffner. Learning generalized policies without supervision using gnns. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning (KR'22)*, 2022.

[32] M. Steinmetz, D. Fišer, H. F. Enişer, P. Ferber, T. Gros, P. Heim, D. Höller, X. Schuler, V. Wüstholz, M. Christakis, and J. Hoffmann. Debugging a policy: Automatic action-policy testing in AI planning. In ICAPS-22 [19].

[33] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[34] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[35] S. Toyer, S. Thiébaux, F. W. Trevizan, and L. Xie. Asnets: Deep learning for generalised planning. *Journal of Artificial Intelligence Research*, 68:1–68, 2020.

[36] M. Vinzent, M. Steinmetz, and J. Hoffmann. Neural network action policy verification via predicate abstraction. In ICAPS-22 [19].

[37] W. Zhao, T. He, R. Chen, T. Wei, and C. Liu. State-wise safe reinforcement learning: A survey. In E. Elkind, editor,

Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23, pages 6814–6822. International Joint Conferences on Artificial Intelligence Organization, 8 2023. Survey Track.