Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)

Policy Comparison Oracles for Action Policy Testing

Ben Sievers[1], **Jan Eisenhut**[1], **Jörg Hoffmann**[1,2]

[1]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2]German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
besi00002@stud.uni-saarland.de, eisenhut@cs.uni-saarland.de, hoffmann@cs.uni-saarland.de

Abstract

Testing is a natural quality assurance technique for learned action policies π. In classical planning, the testing process attempts to find states, called bugs, on which the plan generated by π is sub-optimal. A major challenge in this context is the design of test oracles, sufficient criteria for identifying bugs. Here, we introduce a new type of such oracles, that we call policy comparison oracles (PCOs). These are based on comparing π with a set of policies π' produced during the training process for π. Trivially, on a given state s, if any π' is better than π, then s is a bug in π. But the potential of policy comparison reaches far beyond that. For example, π' may produce a better sub-plan on s even if it does not reach the goal at all. We introduce a combination method that allows to arbitrarily alternate between policies at testing time, thus leveraging their combined potential. We run experiments using ASNets policies. PCOs turn out to be competitive with state-of-the-art test oracles on their own, and their integration with other oracles is superior in our evaluation.

Source code and benchmarks —
github.com/fai-saarland/bughive/tree/icaps26-pcos

Introduction

Learned action policies π are gaining ever more traction in AI planning (e.g., Groshev et al. 2018; Garg, Bajpai, and Mausam 2019; Toyer et al. 2020; Ståhlberg, Bonet, and Geffner 2022; Wang and Thiébaux 2024; Rossetti et al. 2024). Once trained, they can be valuable tools for decision making under real-time constraints, scaling beyond traditional planning systems. Yet, they are often unreliable, making the development of efficient quality assurance methods a necessity. To that end, prior work (Steinmetz et al. 2022; Eisenhut et al. 2023, 2024) introduced policy testing as a means for finding "bugs" in action policies. A state s is called a bug in π, if π performs sub-optimally on s, either by not reaching the goal even though s is solvable, or reaching it with unnecessarily high cost. Testing is organized as a two-step process: first, random walks are used to generate a pool of test states; then sufficient criteria, so-called test oracles, are run on the test pool to identify bugs (avoiding the need for optimal planning whose scalability is limited).

In this work, we introduce a new class of test oracles that we baptize policy comparison oracles (PCOs). Our basic observations are that 1. a simple method to assess the quality of policy π is to compare it with another policy π'; 2. as training processes usually iterate through many intermediate policies π', we get an entire set of alternative policies for free. Trivially, if a π' is better on a state s than π, then s is a bug in π. But policy comparison has far greater potential. For example, π' may be worse than π on s itself, but produce a better sub-plan for an intermediate state t in π's run. Even if π' does not reach the goal from s, it may contain a better plan from s to such a t. In both cases, a suitably combined policy run beats both π and π'.

Specifically, we introduce a combination method that allows to arbitrarily alternate between policies, by viewing them all as sources of upper bounds on $h^\star$ in a search space explored at policy testing time. To realize this, we build on Eisenhut et al.'s (2023) bound-maintenance oracles (BMOs), which are based on comparing multiple states in the same policy. We contribute methods that compare multiple policies on the same states and directly integrate them into existing BMOs to enable synergies.

While calling learned policies is polynomial time in the size of the model, policy representations can be large. Thus, the time required for running policies is still a bottleneck in policy testing. In particular, there is a practical trade-off between the number of states we can test and the number of policies we can run on each state. To decide which policies should be tried on which states, we consider how to inform policy selection by keeping track of policy behavior.

We run experiments on benchmarks of the IPC 2023 learning track, using ASNets policies (Toyer et al. 2020) from (Eisenhut et al. 2025). We first compare coverage and plan quality of a single policy against the best-of over a set of policies. We observe significant complementarity and thus improved overall performance when using the polices as a trivial portfolio (running each policy in turn). This highlights the potential of PCOs; it also provides first positive results for policy portfolio design, exploring which in depth remains a topic for future work (the different challenge being to satisfy real-time constraints). In our main experiment, we compare PCOs with state-of-the-art test oracles (of Eisenhut et al. 2023). We observe that PCOs are competitive and their integration with other oracles is superior in our evaluation.

Background

We first provide background on finite-domain representation (FDR) planning, action policies and policy testing.

An FDR **task** P is a tuple $\langle \mathcal{V}, \mathcal{A}, I, G \rangle$. $\mathcal{V}$ is a finite set of **variables** v, each with finite domain $\mathrm{dom}(v)$. We call partial variable assignments over $\mathcal{V}$ **partial states** and full assignments **states**. $\mathcal{S}$ is the set of all states, $I \in \mathcal{S}$ is an **initial state**, G is a partial state called **goal**, and a state s is a **goal state** if $G \subseteq s$. $\mathcal{A}$ is a finite set of **actions** a of the form $\langle \mathrm{pre}(a), \mathrm{eff}(a), c(a) \rangle$, with **precondition** $\mathrm{pre}(a)$ and **effect** $\mathrm{eff}(a)$, both partial states, and **cost** $c(a) \in \mathbb{R}_0^+$. An action a is **applicable** in a state s if $\mathrm{pre}(a) \subseteq s$ and $\mathcal{A}(s)$ is the set of all actions applicable in s. Applying $a \in \mathcal{A}(s)$ in s results in a state $s[\![a]\!]$ obtained by copying s and assigning all variables v affected by $\mathrm{eff}(a)$ to $\mathrm{eff}(a)[v]$. An action *sequence* $\vec{a} = \langle a_1, \ldots, a_n \rangle$ is applicable in a state s_0 if there exist states $s_1, \ldots, s_n$ with $a_i \in \mathcal{A}(s_{i-1})$ and $s_i = s_{i-1}[\![a_i]\!]$. Applying $\vec{a}$ in s_0 results in $s_0[\![\vec{a}]\!] = s_n$. If $\vec{a}$ is applicable in s and $s[\![\vec{a}]\!]$ is a goal state, $\vec{a}$ is an s-**plan** with cost $c(\vec{a}) = \sum_{i=1}^{n} c(a_i)$. The **cost-to-goal** function $h^\star$ maps each state s to the cost of a cheapest s-plan, or to ∞ if none exists.

Deterministic **action policies** $\pi \colon \mathcal{S} \to \mathcal{A} \cup \{\bot\}$ map states s to actions $a \in \mathcal{A}(s)$, or $\bot$ if $\mathcal{A}(s) = \emptyset$. We run π on s by iteratively applying π until we reach the goal, π selects $\bot$, or we run into a loop. Formally, the **run** $\sigma^\pi(s)$ of π on $s = s_0$ is the longest sequence $\langle s_0, a_0, \ldots, a_{n-1}, s_n \rangle$ of actions a_i and pairwise different states s_i such that $a_i = \pi(s_i) \in \mathcal{A}(s_i)$, $s_{i+1} = s_i[\![a_i]\!]$, and $G \not\subseteq s_i$ for all $i < n$. If $\sigma^\pi(s)$ ends in a goal state, π **solves** s with **cost** $c^\pi(s) = c(\langle a_0, \ldots, a_{n-1} \rangle)$. Otherwise, $c^\pi(s)$ is ∞. While our methods are generic, we focus on learned action policies, specifically ASNets (Toyer et al. 2018, 2020), which are domain-generalized. Since the training process iterates through many intermediate policies, we learn a set of policies rather than a single one. As an alternative to simply selecting the result of the final training step, Eisenhut et al. (2025) propose more sophisticated ranking techniques based on policy testing on validation tasks.

In Steinmetz et al.'s (2022) testing framework, a state s is a **bug** in policy π if $c^\pi(s) > h^\star(s)$. We call s a **qualitative bug** if π does not solve s even though $h^\star(s) < \infty$. Otherwise, if π solves s but there exists a cheaper plan, we call s a **quantitative bug**. Testing encompasses two (possibly interleaved) steps: building a pool $\mathcal{P} \subseteq \mathcal{S}$ of test states, and invoking **test oracles** on them, sufficient criteria for identifying bugs. In this work, we only consider test oracles. The best available oracles so far combine several methods. A central paradigm (proposed in Eisenhut et al. 2023) is to share information across oracle calls, instead of treating each as a separate problem. This can be realized by maintaining bounds $u[s] \geq h^\star(s)$ across the states s encountered during testing. Clearly, if we find a bound $u[s] < c^\pi(s)$, s is a bug in π. To propagate bounds, Eisenhut et al.'s (2023) bound-maintenance oracles (BMOs) leverage metamorphic testing techniques, i.e., bounds are propagated by comparing states via dominance functions (Torralba 2017). In addition to policy runs as obvious source of bounds, any method can be integrated, e.g., local search techniques ("lookahead search").

Policy Comparison Oracles

Our new policy comparison oracles (PCOs) aim at identifying bugs in a policy by comparing it to alternative ones. Throughout this section, we fix a policy under test π and a portfolio $\Pi = \{\pi_1, \ldots, \pi_n\}$ of secondary policies. While our PCOs are generic, we focus on the case where all policies are based on the same architecture and stem from the same training process. This facilitates ranking the policies based on training or validation data. In our setting, we choose 20 portfolio policies based on such a ranking.

The simplest possible PCO only attempts to find bugs in π by running individual portfolio policies $\pi_i \in \Pi$. Clearly, if any π_i is better than π on a state s, then π cannot be optimal on s. Formally, if $c^{\pi_i}(s) < c^\pi(s)$, then also $h^\star(s) < c^\pi(s)$ so that s is a bug in π. To check this criterion efficiently, we determine $c^\pi(s)$ first and stop computing the run of a π_i if we can rule out that π_i is better on s, i.e., if the cost of the generated partial run of π_i reaches $c^\pi(s)$. Moreover, there is obviously no need to run further portfolio policies once we have identified s as a bug in π.

Dynamic Policy Selection

To keep runtime at bay, we require a method to limit the number of policy runs per state. The most obvious way to achieve this would be to reduce the size of our portfolio Π, completely ignoring policies. This however is problematic for various reasons. First, pre-computed policy rankings based on validation data can only serve as guides for actual quality. Second, even assembling the individually best policies does not guarantee best collective performance, e.g., it can be expedient to include policies despite poor average performance if they work well in distinct regions of the state space. For this reason, we choose a generous portfolio size and consider how to dynamically select portfolio policies.

Algorithm 1: PCO with dynamic policy selection

```
1  B ← {(π_i, 0) | π_i ∈ Π}; N ← {(π_i, 0) | π_i ∈ Π};
2  R ← {(π_i, 1) | π_i ∈ Π};
3  Procedure PCO_P(s, Π, m)
4      Q ← Π;                                   // still untried policies
5      for j = 1, ..., min(m, |Π|) do
6          π_i ← some π_k ∈ Q with R[π_k] = max_{π'∈Q} R[π'];
7          Q ← Q \ {π_i};
8          N[π_i] ← N[π_i] + 1; R(π_i) ← B[π_i]/N[π_i];
9          if c^{π_i}(s) < c^π(s) then
10             B[π_i] ← B[π_i] + 1; R[π_i] ← B[π_i]/N[π_i];
11             flagBug(s, π); return;
```

To inform policy selection, we track the success of each π_i in finding bugs in π. Algorithm 1 shows pseudo-code for this oracle $\mathrm{PCO_P}$. We try up to m portfolio policies π_i per test state s, checking $c^{\pi_i}(s) < c^\pi(s)$. Obviously, $c^\pi(s)$ only needs to be computed once. In each iteration, we select π_i with maximal bugs ratio $R[\pi_i] = B[\pi_i]/N[\pi_i]$, where $B[\pi_i]$ is the number of bugs in π found by π_i and $N[\pi_i]$ is the total number of π_i's runs. B, N, and R are maintained across oracle invocations. We ensure that each π_i is tried at most

once per state (using variable Q). To increase the chance that each portfolio policy π_i is tried at least for one test state, we initialize $R[\pi_i]$ with 1, the highest possible value for R.

Combining and Abstracting from Policies

We now explore ways to *combine* policies rather than merely comparing the cost of separate runs. As a first approach, we compare π and portfolio policies π_i on intermediate states s_j in the run $\sigma^\pi(s) = \langle s_0, a_0, s_1, \ldots, a_{n-1}, s_n \rangle$ from $s = s_0$. If $c^{\pi_i}(s_j) < c^\pi(s_j)$, then the plan combining π from s to s_j and π_i from s_j to the goal is cheaper than $c^\pi(s)$. Yet, to show that s is a bug in π, it is not necessary to explicitly construct such combined runs; it suffices to know that s_j is a bug in π and that s_j is in π's run. Specifically, if s_j is a bug in π, then $h^\star(s_j) < c^\pi(s_j)$, which implies $h^\star(s_{j-1}) \leq h^\star(s_j) + c(a_{j-1}) < c^\pi(s_j) + c(a_{j-1}) = c^\pi(s_{j-1})$, so that s_{j-1} is also a bug in π and, by iteration, so is s.

Algorithm 2: Extended PCO

```
Procedure PCO_E(s, Π, m, p)
    b ← ⊥; ⟨s_0, a_0, ..., s_{n-1}, a_{n-1}, s_n⟩ ← σ^π(s);
    for j = n, ..., 0 do
        if b then flagBug(s_j, π);
        if ¬b ∧ random(p) then
            PCO_P(s_j, Π, m); b ← isFlaggedBug(s_j, π);
```

Algorithm 2 shows pseudo-code for this oracle PCO_E. We iterate through the states s_j in $\sigma^\pi(s)$ backwards and invoke PCO_P to check for cheaper paths from s_j. In addition, we always check whether s_j has been flagged as a bug in π and, if so, flag all previous states in $\sigma^\pi(s)$ as bugs in π. As a means to control runtime, we add a parameter p, denoting the probability of the PCO_P call being executed. A benefit of iterating through $\sigma^\pi(s)$ backwards are potentially shorter auxiliary policy runs, particularly if π reached the goal.

Abstracting from Policies by Maintaining Bounds

To unlock more collective potential, we propose a method that integrates data from multiple policies via abstraction from individual runs. Following Eisenhut et al. (2023), we maintain bounds $u[t] \geq h^\star(t)$ for all generated states t. In particular, whenever running a policy, we attempt to obtain upper bounds for all intermediate states t and check whether we can lower $u[t]$. If $c^\pi(t)$ is already known, we also check $u[t] < c^\pi(t)$ to see if we can flag t as a bug in π.

In addition, we may propagate such bounds to ancestors: Given a state t, an action $a \in \mathcal{A}(t)$ with successor $t' = t[\![a]\!]$, and a bound $u[t'] \geq h^\star(t')$, we have $u[t'] + c(a) \geq h^\star(t)$. We can use this to propagate along reversed policy runs. For example, if π solves a state s but π_i finds a better sub-path from s to an intermediate state s_j in π's run, propagating $u[s_j]$ backwards along π_i's run can confirm s as a bug in π.

To add bound propagation to PCO_E, we store the predecessors s_j and generating actions a for the states s_{j+1} visited in any policy runs. Each $u[t]$ update then triggers a backwards traversal of t's ancestor states, updating their bounds if possible. We omit pseudo code here since the procedure is straightforward and very similar to Eisenhut et al.'s (2023) "updateAncestors" optimization.

Taking this one step further, we can also directly integrate our method into Eisenhut et al.'s (2023) BMOs, gaining access to their bound propagation techniques and optimizations. We can simply share upper bounds between a PCO and a BMO, making sure that whenever either oracle updates a bound, this new bound is immediately available to the other one. From the perspective of the BMO, a PCO primarily serves as an external initial source for cost bounds. However, PCOs can also contribute improved backwards bound propagation, as the registered predecessor states are not only based on π alone, but on a portfolio of policies. Our implementation of combined oracles calls the BMO first. Only if the BMO cannot identify a state as a bug, the PCO is called. Our rationale for this is that BMO calls are often faster, since they do not require additional policy runs.

Domain	$\#P$	Coverage				Average plan cost			
		1	2	5	10	1	2	5	10
Blocksworld	60	51	55	**56**	**56**	189.2	189.1	186.8	**186.2**
Childsnack	60	42	48	52	**53**	54.9	54.8	54.0	**52.5**
Ferry	63	**63**	**63**	**63**	**63**	130.4	**130.0**	**130.0**	**130.0**
Floortile	60	22	24	**33**	**33**	55.6	55.3	**55.0**	**55.0**
Miconic	70	**70**	**70**	**70**	**70**	119.8	118.6	116.0	**115.7**
Rovers	39	12	17	17	**19**	28.0	28.0	**27.2**	**27.2**
Satellite	60	58	**60**	**60**	**60**	58.6	56.6	54.4	**54.2**
Sokoban	43	7	**8**	**8**	**8**	**12.1**	**12.1**	**12.1**	**12.1**
Spanner	84	81	**82**	**82**	**82**	159.8	159.8	159.8	**159.2**
Transport	55	42	43	44	**45**	36.2	36.0	35.3	**35.1**

Table 1: Coverage and average plan cost (over commonly solved tasks) for 1, 2, 5, and 10 policies. $\#P$: Domain size.

Experiments

We run experiments for two main objectives: first, to assess the degree of complementarity between the policies in a planning setting, to gauge the general potential of policy combinations, and second, to evaluate the performance of our new oracles in comparison to previous approaches.

To conduct our experiments, we extend Steinmetz et al.'s (2022) policy testing framework building on Fast Downward (Helmert 2006). Policies and PDDL benchmarks are taken from (Eisenhut et al. 2025). Specifically, we employ Fišer's (2025) cpddl implementation of ASNets, to which we add support for serving multiple policies at the same time. We use a set of policies trained by Eisenhut et al. (2025), including 240 (intermediate) policies. As policy under test π, we fix the policy selected by their "test score" criterion. In addition, we choose a ranked list of 20 portfolio policies, based on the validation data they used for policy selection. In more detail, this builds on initial state coverage on a separate validation set, using percentage of solved pool states and testing results as tie-breakers. The PDDL benchmarks are part of the test set of the IPC 2023 learning track, omitting tasks where pre-processing steps (such as grounding as used by ASNets or pre-computations for test oracles) run

Domain	$\#P$	$\lvert T_U \rvert$	% of states in T_U flagged as (qualitative) bugs						$\lvert T_S \rvert$	% of states in T_S flagged as (quantitative) bugs					
			Aras/ EHC	BMO_P	BMO_E	PCO_P	PCO_E	CO		Aras/ EHC	BMO_P	BMO_E	PCO_P	PCO_E	CO
Blocksworld	60	675	27.3	5.3	48.1	34.2	35.4	**75.0**	3983	32.2	0.3	33.4	65.3	68.5	**77.4**
Childsnack	60	2493	2.7	1.2	2.7	2.8	2.8	**3.2**	731	42.4	4.0	42.4	25.9	25.7	**42.8**
Ferry	63	0	–	–	–	–	–	–	4674	6.8	0.3	7.0	15.6	16.2	**17.6**
Floortile	60	2569	0.0	2.1	2.6	7.1	9.3	**9.6**	494	31.6	4.7	31.6	18.2	23.3	**32.4**
Miconic	70	0	–	–	–	–	–	–	5809	15.2	46.8	51.9	57.3	61.2	**63.2**
Rovers	39	1956	98.6	49.7	**99.4**	38.3	44.1	99.2	1169	37.5	31.3	37.5	25.3	29.3	**37.7**
Satellite	60	65	64.6	10.8	96.9	**100.0**	**100.0**	**100.0**	3700	32.1	18.1	47.0	35.0	36.1	**50.9**
Sokoban	43	3076	60.1	10.5	**73.2**	5.1	6.1	**73.2**	600	**28.5**	28.2	**28.5**	4.8	5.3	**28.5**
Spanner	84	611	0.0	3.6	3.6	4.4	4.4	**4.9**	4895	79.3	55.4	79.4	77.4	78.6	**79.8**
Transport	55	282	64.9	91.5	98.6	85.5	95.7	**99.3**	3393	30.1	16.7	36.9	42.0	47.1	**56.3**

Table 2: Percentage of pool states flagged as bugs, distinguishing qualitative (left) and quantitative ones (right). T_S (T_U) is the domain-wide set of test states (not) solved by π, only including states tested by all oracles. $\#P$ is the number of tasks.

out of memory. All experiments were run on a cluster of Intel E5-2660 processors running at 2.20 GHz with a memory limit of 8 GiB and a time limit of two hours per task.

Evaluation of Policy Complementarity

To assess the complementarity of the policies at hand, we first compare initial state coverage and plan quality for a single policy to policy portfolios of different sizes. Specifically, the portfolio of size m includes π and the best ranked $m-1$ auxiliary policies. We run each policy in turn, stopping in case we reach the (portfolio wide) two hour time limit. We then choose the best plan achieved by any policy.

Table 1 shows the results for coverage and average plan cost. We observe highly complementary behavior in some domains. For coverage, this is particularly pronounced in Childsnack and Floortile, where the portfolios of ten policies solve eleven more instances than π alone. At the same time, average plan cost (for tasks solved by all configurations) decreases marginally. In general, our results indicate potential in combining multiple policies in more sophisticated ways, also encouraging the application in test oracles.

Evaluation of Policy Comparison Oracles

We compare six different oracles. As baseline, this includes the Aras/EHC, BMO_P and BMO_E oracles from previous work (Steinmetz et al. 2022; Eisenhut et al. 2023). The first oracle uses the Aras plan improvement tool (Nakhost and Müller 2010), given a plan is found initially, and Enforced Hill Climbing (EHC) with h^{FF} (Hoffmann and Nebel 2001) otherwise. The last two oracles employ metamorphic approaches: BMO_P only uses metamorphic comparisons. BMO_E additionally uses lookahead search (up to depth 100) and employs the Aras/EHC oracle as a further source of upper bounds. This oracle showed best performance in previous work (Eisenhut et al. 2023).

Our new policy comparison oracles, PCO_P and PCO_E, both run $m = 5$ policies per pool state. PCO_P keeps track of previous bug finding performance to inform policy selection from the portfolio. PCO_E additionally starts portfolio runs along the main policy's path (one run with $p = 30\%$ probability per path state) and uses the described bound maintenance extension. The composite oracle (CO) combines metamorphic and policy comparison approaches using BMO_E and PCO_E (and hence also Aras/EHC): If the former does not flag the pool state as a bug, the latter is invoked on the same pool state. The oracles are integrated in that they share upper bounds on $h^\star$ to enable synergies, i.e., BMO_E can leverage bounds derived in PCO_E calls and vice versa.

We call each oracle on a maximum of 100 pool states per task, using pre-computed pools from (Eisenhut et al. 2025). Table 2 shows the results, distinguishing qualitative and quantitative bug finding performance. T_S and T_U are the pool states (across all tasks) that are solved or, respectively, not solved by the policy under test π. In both cases, we only include states processed by all compared oracles. The left side of Table 2 shows the percentage of (unsolved) states in T_U flagged as (qualitative) bugs; the right side the percentage of (solved) states in T_S flagged as (quantitative) bugs.

We observe that PCO_P is competitive on its own, beating BMO_E in several domains, both for detecting qualitative and quantitative bugs. In addition, the more advanced PCO_E configuration improves on PCO_P in most domains. The composite oracle CO strictly dominates its components in almost all domains. This, in particular, highlights the synergies and complementary behavior between its components, and yields the most powerful oracle in our evaluation.

Conclusion

The proliferation of learned action policies π necessitates the development of effective testing techniques, particularly test oracles capable of efficiently detecting bugs. Here, we contribute a novel class of test oracles based on comparing and combining π with auxiliary policies. In our evaluation using ASNets policies in classical planning, these show competitive performance on their own and their integration with existing methods advances the state-of-the-art. Motivated by our observation that these policies exhibit a significant degree of complementarity in our experiments, we believe that the idea of combining policies has inherent potential not just in policy testing, but in planning more widely.

Acknowledgments

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

References

Eisenhut, J.; Fišer, D.; Valera, I.; and Hoffmann, J. 2025. On Picking Good Policies: Leveraging Action-Policy Testing in Policy Training. In *Proceedings of the 35th International Conference on Automated Planning and Scheduling (ICAPS'25)*, 183–188. AAAI Press.

Eisenhut, J.; Schuler, X.; Fišer, D.; Höller, D.; Christakis, M.; and Hoffmann, J. 2024. New Fuzzing Biases for Action Policy Testing. In *Proceedings of the 34th International Conference on Automated Planning and Scheduling (ICAPS'24)*, 162–167. AAAI Press.

Eisenhut, J.; Torralba, Á.; Christakis, M.; and Hoffmann, J. 2023. Automatic Metamorphic Test Oracles for Action-Policy Testing. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling (ICAPS'23)*, 109–117. AAAI Press.

Fišer, D. 2025. cpddl. Zenodo. https://doi.org/10.5281/zenodo.16423302.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS'19)*, 631–636. AAAI Press.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS'18)*, 408–416. AAAI Press.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Nakhost, H.; and Müller, M. 2010. Action Elimination and Plan Neighborhood Graph Search: Two Algorithms for Plan Improvement. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS'10)*, 121–128. AAAI press.

Rossetti, N.; Tummolo, M.; Gerevini, A. E.; Putelli, L.; Serina, I.; Chiari, M.; and Olivato, M. 2024. Learning General Policies for Planning through GPT Models. In *Proceedings of the 34th International Conference on Automated Planning and Scheduling (ICAPS'24)*, 500–508. AAAI Press.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling (ICAPS'22)*, 629–637. AAAI Press.

Steinmetz, M.; Fišer, D.; Enişer, H. F.; Ferber, P.; Gros, T.; Heim, P.; Höller, D.; Schuler, X.; Wüstholz, V.; Christakis, M.; and Hoffmann, J. 2022. Debugging a Policy: Automatic Action-Policy Testing in AI Planning. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling (ICAPS'22)*, 353–361. AAAI Press.

Torralba, Á. 2017. From Qualitative to Quantitative Dominance Pruning for Optimal Planning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 4426–4432.

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

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies with Deep Learning. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*, 6294–6301. AAAI Press.

Wang, R. X.; and Thiébaux, S. 2024. Learning Generalised Policies for Numeric Planning. In *Proceedings of the 34th International Conference on Automated Planning and Scheduling (ICAPS'24)*, 633–642. AAAI Press.