Neural Action Policy Safety Verification: Applicablity Filtering Technical Report

Marcel Vinzent[1], **Jörg Hoffmann**[1,2]

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

Abstract

Neural networks (NN) are an increasingly important representation of action policies π. Applicability filtering is a commonly used practice in this context, restricting the action selection in π to only applicable actions. Policy predicate abstraction (PPA) has recently been introduced to verify safety of neural π, through over-approximating the state space subgraph induced by π. Thus far however, PPA does not permit applicability filtering, which is challenging due to the additional constraints that need to be taken into account. Here we overcome that limitation, through a range of algorithmic enhancements. In our experiments, our enhancements achieve several orders of magnitude speed-up over a baseline implementation, bringing PPA with applicability filtering close to the performance of PPA without such filtering.

1 Introduction

Neural networks (NN) are an increasingly important representation of action policies in many contexts, including AI planning (Issakkimuthu, Fern, and Tadepalli 2018; Groshev et al. 2018; Garg, Bajpai, and Mausam 2019). But how to verify that such a *policy* π is safe? Given a *start condition* ϕ_0 and an *unsafety condition* ϕ_u, how to verify whether an unsafe state $s^u \models \phi_u$ is reachable from a start state $s^0 \models \phi_0$ under π? Such verification is potentially very hard as it compounds the state space explosion problem with the difficulty of analyzing even single NN decision episodes. A prominent line of work addresses neural controllers of dynamical systems, where the NN output forms input to a continuous state-evolution function (Tran et al. 2019; Huang et al. 2019; Dutta, Chen, and Sankaranarayanan 2019; Ivanov et al. 2021). A recent thread explores bounded-length verification of neural controllers (Akintunde et al. 2018, 2019; Amir, Schapira, and Katz 2021).

Here we follow up on work on *policy predicate abstraction* (PPA) by Vinzent et al. (2022; 2023) (henceforth: *VEA*), which tackles neural policies π that take discrete action choices in non-deterministic state spaces. Like classical predicate abstraction (Graf and Saïdi 1997), PPA builds an over-approximating abstraction defined through a set $\mathcal{P}$ of *predicates*, i.e., linear constraints over the state variables. However, PPA abstracts not the full state space, but the subgraph induced by π. To compute the abstract state space $\Theta^{\pi}_{\mathcal{P}}$, one must repeatedly solve the sub-problem of deciding whether there is a transition from abstract state $s_{\mathcal{P}}$ to abstract state $s'_{\mathcal{P}}$ under π. This *abstract transition problem* is encoded into satisfiability modulo theories (SMT) (Barrett and Tinelli 2018), and answered querying solvers tailored to NN analysis (Katz et al. 2019). If there does not exist a path from ϕ_0 to ϕ_u in $\Theta^{\pi}_{\mathcal{P}}$, then π is safe. Counterexample-guided abstraction refinement (CEGAR) (Clarke et al. 2003) is deployed to iteratively refine $\mathcal{P}$ until either π is proven safe or an unsafe counterexample is found. In an empirical evaluation, VEA show that their approach outperforms encodings into the state-of-the-art verification tool NUXMV (Cavada et al. 2014).

VEA consider neural policies that may select any action in any state, including *inapplicable* actions. This makes it unnecessarily difficult to learn good policies. Instead, an established practice is to *filter* the selection of π with respect to *applicability* (Toyer et al. 2020; Stahlberg, Bonet, and Geffner 2022). On the verification side, however, applicability filtering is challenging since it introduces additional disjunctive behavior into the abstract transition problem: π *may select action label l depending on whether another action l' is or is not applicable.* Implemented straightforwardly, PPA with applicability filtering suffers from a huge performance loss. In our experiments on VEA's benchmarks, it runs out of time or memory on all but the smallest instances – which, without applicability filtering, PPA tackles in a few seconds. In this paper, we devise a range of algorithmic enhancements that overcome this limitation. The enhancements exploit SMT-solver-specific encoding strategies, and simplify disjunctions in the SMT encoding of the applicability filter based on entailment of sub-constraints. Empirically, these methods achieve runtime improvements of up to three orders of magnitude, and bring PPA with applicability filtering close to the performance of PPA without such filtering.

We also refine VEA's notion of safety, in that we consider the more accurate reach-avoid setting where the task of the learned policy is to reach a goal state while avoiding unsafe states. Policy executions in reach-avoid stop at goal states. In VEA's prior work, a policy can, at least in principle, be unsafe even though unsafe states are encountered only after reaching the goal.

2 Preliminaries

We consider discrete non-deterministic transition systems described by a tuple $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$ where $\mathcal{V}$ is a finite set of bounded-integer *state variables*, $\mathcal{L}$ is a finite set of *action labels* and $\mathcal{O}$ is a finite set of *operators*. We denote by Exp the set of *linear expressions* over $\mathcal{V}$, i.e., of the form $\sum_{v \in \mathcal{V}} d_v \cdot v + c$ with coefficients $d_v \in \mathbb{Z}$ for each $v \in \mathcal{V}$ and $c \in \mathbb{Z}$. Accordingly, C denotes the set of *linear constraints*, of the form $\sum_{v \in \mathcal{V}} d_v \cdot v \geq c$, and Boolean combinations thereof.[1] An *operator* $o \in \mathcal{O}$ is a tuple (l, g, u) with *label* $l \in \mathcal{L}$, *guard* $g \in C$ (a conjunction of linear constraints), and (linear) *update* $u \colon \mathcal{V} \to Exp$.

The *state space* of $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$ is a labeled transition system $\Theta = \langle \mathcal{S}, \mathcal{L}, \mathcal{T} \rangle$. The set of *states* $\mathcal{S}$ is the finite set of all complete variable assignments over $\mathcal{V}$. The set of *transitions* $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ contains (s, l, s') iff there exists an operator $o = (l, g, u)$ such that g is satisfied over s, formally $g(s)$ evaluates to true, also abbreviated $s \models o$, and $s'(v)$ maps to the update $u(v)$ evaluated over s for each $v \in \mathcal{V}$, formally $s' = \{v \mapsto u(v)(s) \mid v \in \mathcal{V}\}$, also abbreviated $s' = s[\![o]\!]$. The separation between action labels and operators allows both, state-dependent effects (different operators with the same label l applicable in different states) and action outcome non-determinism (different operators with the same label l applicable in the same state).

An *action policy* π is a function $\mathcal{S} \to \mathcal{L}$. We consider π represented by a *neural network* (NN). Specifically, we focus on feed-forward NN with *rectified linear unit* (ReLU) activations $ReLU(x) = \max(x, 0)$. These NN consist of an input layer, arbitrarily many hidden layers, and an output layer with one neuron per label $l \in \mathcal{L}$. A *safety property* is a pair (ϕ_0, ϕ_u), where $\phi_0 \in C$ and $\phi_u \in C$ identify the set of start and unsafe states respectively. A policy π is *unsafe* with respect to (ϕ_0, ϕ_u) iff there exists a state path $\langle s^0, \ldots, s^n \rangle$ such that $s^0 \models \phi_0$, $s^n \models \phi_u$, and $(s^i, \pi(s^i), s^{i+1}) \in \mathcal{T}$ for $i \in \{0, \ldots, n-1\}$. Otherwise π is *safe*.

Policy predicate abstraction (PPA) (Vinzent, Steinmetz, and Hoffmann 2022) is an extension of classical predicate abstraction (Graf and Saïdi 1997). Unlike its classical counterpart, PPA abstracts not the full state space, but the subgraph induced by π. Assume a set of *predicates* $\mathcal{P} \subseteq C$. An *abstract state* $s_{\mathcal{P}}$ is a complete truth value assignment over $\mathcal{P}$. $[s_{\mathcal{P}}] = \{s \in \mathcal{S} \mid \forall p \in \mathcal{P} \colon p(s) = s_{\mathcal{P}}(p)\}$ denotes the set of concrete states represented by $s_{\mathcal{P}}$. The *policy predicate abstraction* of Θ over $\mathcal{P}$ and π is the labeled transition system $\Theta^{\pi}_{\mathcal{P}} = \langle \mathcal{S}_{\mathcal{P}}, \mathcal{L}, \mathcal{T}^{\pi}_{\mathcal{P}} \rangle$ where $\mathcal{S}_{\mathcal{P}}$ is the set of abstract states over $\mathcal{P}$ and $(s_{\mathcal{P}}, l, s'_{\mathcal{P}}) \in \mathcal{T}^{\pi}_{\mathcal{P}}$ iff there exists $(s, l, s') \in \mathcal{T}$ such that $s \in [s_{\mathcal{P}}]$, $s' \in [s'_{\mathcal{P}}]$ and $\pi(s) = l$.

Analogously to safety in Θ, π is said to be *unsafe* in $\Theta^{\pi}_{\mathcal{P}}$ iff there exists an abstract path $\langle s^0_{\mathcal{P}}, l^0, \ldots, l^{n-1}, s^n_{\mathcal{P}} \rangle$ such that $s^0 \models \phi_0$ for some $s^0 \in [s^0_{\mathcal{P}}]$, $s^n \models \phi_u$ for some $s^n \in [s^n_{\mathcal{P}}]$, and $(s^i_{\mathcal{P}}, l^i, s^{i+1}_{\mathcal{P}}) \in \mathcal{T}^{\pi}_{\mathcal{P}}$ for $i \in \{0, \ldots, n-1\}$. Otherwise π is *safe* in $\Theta^{\pi}_{\mathcal{P}}$, in which case it is safe in Θ as well. An (unsafe) abstract path in $\Theta^{\pi}_{\mathcal{P}}$ may be *spurious*, i.e., there does not exist a corresponding path in Θ under π. *Counterexample-guided abstraction refinement* (CEGAR) (Clarke et al. 2003) iteratively removes such spurious abstract paths by refining $\mathcal{P}$, until either the abstraction is proven safe, or a non-spurious abstract path is found proving π unsafe. VEA provide a CEGAR framework specialized to PPA (Vinzent, Sharma, and Hoffmann 2023).

To compute $\Theta^{\pi}_{\mathcal{P}}$, one must solve the **abstract transition problem** for every possible abstract transition: $(s_{\mathcal{P}}, l, s'_{\mathcal{P}}) \in \mathcal{T}^{\pi}_{\mathcal{P}}$ iff for some l-labeled operator $o \in \mathcal{O}$ there exists a concrete state $s \in [s_{\mathcal{P}}]$ such that $s \models o$, $s[\![o]\!] \in [s'_{\mathcal{P}}]$ and $\pi(s) = l$. In the classical setting where no policy is considered and thus condition $\pi(s) = l$ is not needed, such abstract transition problems are routinely encoded into satisfiability modulo theories (SMT) (e.g. (Barrett and Tinelli 2018)). For PPA however, the policy condition $\pi(s) = l$ introduces a key new source of complexity as the SMT sub-formula representing the neural network π contains one non-linear constraint for every ReLU activation. VEA show how this can be dealt with through approximate SMT checks – embedded into an exact decision procedure. In particular, they use continuous relaxations of the bounded-integer state variables, which can be dispatched to *Marabou* (Katz et al. 2019), an SMT solver tailored to NN analysis.

3 Applicability Filtering

VEA consider neural action policies that are obtained by applying argmax to the output of the NN. Let $\pi_l(s)$ be the NN output for label $l \in \mathcal{L}$ given input state $s \in \mathcal{S}$, then $\pi(s) = \operatorname{argmax}_{l \in \mathcal{L}} \pi_l(s)$. Such π may select any label in any state, even if it is not *applicable*, i.e., there does not exist $s' \in \mathcal{S}$ such that $(s, l, s') \in \mathcal{T}$, or equivalently, there does not exist an l-labeled operator o with $s \models o$.

From a learning perspective, allowing π to select inapplicable actions is unnecessarily difficult, as π must learn which actions are applicable in which state. A simple commonplace technique to avoid this is to *filter* the selection of π with respect to *applicability* (e.g. (Toyer et al. 2020)). Formally, the policy under applicability filtering is defined

$$\pi(s) = \begin{cases} \underset{\{l \in \mathcal{L} \mid \exists o \in \mathcal{O}_l \colon s \models o\}}{\operatorname{argmax}} \pi_l(s) & \text{if } \exists o \in \mathcal{O} \colon s \models o \\ \tau & \text{otherwise} \end{cases}$$

where $\mathcal{O}_l = \{(l, g, u) \in \mathcal{O}\}$ is the sef of l-labeled operators and $\tau \in l$ is a *noop*-label $\tau \in \mathcal{L}$ with $\mathcal{O}_\tau = \emptyset$, which we define to be selected iff s is terminal and argmax is undefined (cf. Section 5).

From a verification perspective, applicability filtering also is desirable because, without such filtering, a policy may be safe simply because of *stalling*, selecting an inapplicable action which ends policy execution. However, applicability filtering adds an additional source of complexity to the abstract transition problem, specifically to the policy condition $\pi(s) = l$. In what follows, we focus on the SMT encoding of this sub-problem. The encoding of the neural network itself remains unaffected. We provide a full specification of the SMT encoding in the appendix.

[1] Support extends to "$\leq$" and "$=$" in a straight-forward manner.

Let π_l denote the SMT variable representing the NN output of label l. Without filtering, the policy selection condition is a simple conjunction $\bigwedge_{l' \in \mathcal{L}\setminus\{l\}} \pi_l > \pi_{l'}$. Under applicability filtering however, each conjunct here becomes a disjunction $\bigwedge_{l' \in \mathcal{L}\setminus\{l\}} (\pi_l > \pi_{l'} \vee \neg \bigvee_{o \in \mathcal{O}_{l'}} g_o)$ where g_o denotes the guard of operator o. In words: either the output value of l is greater than that of l', or l' is not applicable.[2] Since each g_o is a conjunction of linear constraints, the selection condition expands to

$$\bigwedge_{l' \in \mathcal{L}\setminus\{l\}} \left(\pi_l > \pi_{l'} \vee \neg \bigvee_{o \in \mathcal{O}_{l'}} \bigwedge_{i \in \{1,\dots,m\}} g_o^i \right)$$

where *sub-guard* g_o^i denotes the i-th linear constraint of guard conjunction g_o and m is the guard size. To simplify notation, we assume that m is constant over all guards – any guard can be extended to some maximal m by adding *trivially-true* constraints.

4 Enhancements

Applicability filtering extends the SMT encoding of the abstract transition problem by a layer of convoluted disjunctions. To tackle this new source of complexity, we devise a range of encoding enhancements that target disjunctions in general and the applicability filter in particular. In the appendix, we provide additional details how these enhancements are deployed as part of VEA's verification algorithm.

Per-operator disjunctions. One type of enhancements exploits the way disjunctions are encoded in *Marabou*, the NN-tailored SMT solver underlying VEA's framework. *Marabou* supports disjunctions in disjunctive normal form (DNF), i.e., $\bigvee_i \bigwedge_j \phi_i^j$ with linear constraints ϕ_i^j. Naively rewriting the top-disjunction $\pi_l > \pi_{l'} \vee \neg \bigvee_{o \in \mathcal{O}_{l'}} \bigwedge_i g_o^i$ into DNF one obtains $\pi_l > \pi_{l'} \vee \bigwedge_{o \in \mathcal{O}_{l'}} \bigvee_i \neg g_o^i$ and then

$$\pi_l > \pi_{l'} \vee \bigvee_{f \in (\mathcal{O}_{l'} \to \{1,\dots,m\})} \bigwedge_{o \in \mathcal{O}_{l'}} \neg g_o^{f(o)}$$

where $\mathcal{O}_{l'} \to \{1,\dots,m\}$ is the set of sub-guard combinations over $\mathcal{O}_{l'}$. Since there are $m^{|\mathcal{O}_{l'}|}$ combinations in total, this encoding is prone to result in a blow-up in size. We overcome this scalability issue by an alternative encoding that splits the top-disjunction into smaller disjunctions

$$\pi_l > \pi_{l'} \vee \bigvee_{i \in \{1,\dots,m\}} \neg g_o^i$$

one for each operator $o \in \mathcal{O}_{l'}$ (PER-OP-DISJ).

Reusing slack variables. *Marabou* transforms every disjunction ϕ to only contain bound tightenings $v \geq c$. Specifically, every non-bound constraint $\sum_{v \in \mathcal{V}} d_v \cdot v \geq c$ in ϕ is transformed to an equation $\sum_{v \in \mathcal{V}} d_v \cdot v + a = c$ where a is a fresh slack variable. This transformed equation is added to the global encoding in a conjunctive manner. The constraint in ϕ is replaced by a bound tightening $a \leq 0$.

We enhance this transformation in that we check for constraints with identical linear combinations $\sum_{v \in \mathcal{V}} d_v \cdot v$ over all disjunctions (OPT-SLACK-VAR). This check detects constraints with multiple occurrences, but also constraints that only differ in the linear offset c. For each such constraint set, we introduce only a single slack variable a, and add a single transformed equation $\sum_{v \in \mathcal{V}} d_v \cdot v + a = 0$ to the global encoding. In all disjunctions, each constraint is replaced by a bound tightening $a \leq -c$, where c is the respective offset of the constraint. In particular, this enhancement pertains to PER-OP-DISJ, where $\pi_l > \pi_{l'}$ occurs multiple times. A formal correctness proof is attached in the appendix.

Entailed sub-constraints. Another type of enhancements exploits *entailment* to simplify the encoding. Given constraints $\psi, \phi \in C$, we say ψ *entails* ϕ, written $\psi \vdash \phi$, iff for every assignment $s \in \mathcal{S}$ such that $s \models \psi$ it also holds $s \models \phi$. Let ϕ be a disjunction $\bigvee_i \bigwedge_j \phi_i^j$ contained in conjunction ψ. If, for some i and j, $\psi \vdash \phi_{\mathrm{i}}^{\mathrm{j}}$, then ϕ can be simplified removing $\phi_{\mathrm{i}}^{\mathrm{j}}$, i.e., $\bigvee_i \bigwedge_{j, i \neq \mathrm{i} \vee j \neq \mathrm{j}} \phi_i^j$. If, for some i, $\psi \vdash \phi_{\mathrm{i}}^j$ for every j, then ψ entails disjunct i and so the entire disjunction ϕ. Hence, ψ can be simplified removing ϕ. If $\psi \vdash \neg\phi_{\mathrm{i}}^{\mathrm{j}}$ for some j, then the entire disjunct i is infeasible and ϕ can be simplified removing i, i.e., $\bigvee_{i \neq \mathrm{i}} \bigwedge_j \phi_i^j$. If all disjuncts i are infeasible, then the entire disjunction ϕ is infeasible and so is ψ. We apply entailment information to optimize the encoding of disjunctions on two levels.

Firstly, on a per operator level (ENTAIL-OP). For each operator o, VEA's algorithm to compute $\Theta_{\mathcal{P}}^{\pi}$ runs an *applicability test* $\exists s \in [s_{\mathcal{P}}]\colon s \models o$. If this test fails then the guard conjunction g_o is entailed to be infeasible in abstract state $s_{\mathcal{P}}$. Say o is l'-labeled. We can use this entailment information to simplify the policy condition for any label $l \neq l'$.

Secondly, on a generic linear level (ENTAIL-GEN) with entailed ϕ in the form of a linear constraint $\sum_{v \in \mathcal{V}} d_v \cdot v \geq c$.

Let $lo_v(\psi)$ and $up_v(\psi)$ denote a lower and upper bound for v entailed by ψ respectively. Then ψ entails ϕ if

$$\sum_{v \in \mathcal{V}^+} d_v \cdot lo_v(\psi) + \sum_{v \in \mathcal{V}^-} d_v \cdot up_v(\psi) \geq c$$

where $\mathcal{V}^+ = \{v \in \mathcal{V} \mid d_v > 0\}$ denotes the variable set with positive coefficients, and $\mathcal{V}^- = \{v \in \mathcal{V} \mid d_v < 0\}$ the variable set with negative coefficients. Analogously, ψ entails $\neg\phi$, we also say ϕ is *infeasible*, if

$$\sum_{v \in \mathcal{V}^+} d_v \cdot up_v(\psi) + \sum_{v \in \mathcal{V}^-} d_v \cdot lo_v(\psi) < c.$$

ψ in the form of the abstract transition problem *syntactically* entails variable bounds in that many predicates in $\mathcal{P}$ are bound constraints $v \geq c$ and, thereby, the conditions $s \in [s_{\mathcal{P}}]$ and $s[\![o]\!] \in [s'_{\mathcal{P}}]$ involve bound tightenings. In addition, Marabou deploys techniques to derive tight bounds on the NN outputs (e.g., (Singh et al. 2019)).

[2] Applicability of l itself is constrained by the full encoding.

The generic entailment check on general linear constraints extends a native check in *Marabou* for infeasible bound constraints in disjunctions – in our experiments the native check is enabled in our baseline.

5 Reach-Avoid Verification

VEA verify safety of π against an unsafety condition ϕ_u given a start condition ϕ_0. However, the task of a practical policy is not only to *avoid* unsafe states, but also to *reach* a goal G – here, a conjunction $\bigwedge_i G_i$ of linear constraints $G_i \in C$. Policy execution stops once G is reached. This corresponds to a *reach-avoid* property: *avoid ϕ_u while not G*. Formally, a policy π is *unsafe* with respect to (ϕ_0, ϕ_u, G) iff there exists a path $\langle s^0, \dots, s^n \rangle$ such that $s^0 \models \phi_0$, $s^n \models \phi_u$, $(s^i, \pi(s^i), s^{i+1}) \in \mathcal{T}$ for $i \in \{0, \dots, n-1\}$ and $s^i \not\models G$ for $i \in \{0, \dots, n\}$. Otherwise π is *safe*.

VEA do not consider reach-avoid, thus potentially reporting unsafe paths that contain goal states. Such counterexamples are not relevant in practice. Hence, we refine VEA's approach to support reach-avoid. Specifically, reach-avoid can be encoded in VEA's framework by annotating the unsafety condition ϕ_u and the guard g of each operator $o \in \mathcal{O}$ with the non-goal condition $\neg G$ – making goal states terminal.

On the algorithmic level, reach-avoid adds another source of complexity to the abstract transition problem, specifically the non-goal disjunction $\bigvee_i \neg G_i$. Our enhancements introduced for applicability filtering can be applied to simplify this disjunction as well. This pertains in particular to ENTAIL-GEN, and ENTAIL-OP with the adapted test $\exists s \in [s_\mathcal{P}]\colon s \models G$. If this test fails, then G is infeasible in $s_\mathcal{P}$.

6 Experiments

We implemented our approach on top of VEA's C++ code base. The enhancements are largely implemented directly into *Marabou*, in particular OPT-SLACK-VAR and ENTAIL-GEN, which is a contribution to improve Marabou's performance on disjunctions in general. All experiments were run on machines with Intel Xenon E5-2650 processors at 2.2 GHz, with time and memory limits of 12 h and 4 GB. Our tool (and all experiments) are publicly available (2024).

Benchmarks. We use VEA's benchmarks. These are non-deterministic variants of the planning domains Blocksworld, SlidingTiles and Transport encoded in JANI (Budde et al. 2017). For each domain instance, there are three NN policies trained by VEA using Q-learning (Mnih et al. 2015), each with two hidden layers of size 16, 32 and 64 respectively, and with ReLU activation nodes. There are policies that do, and ones that do not, take move costs into account.

The policies by VEA are trained without applicability filtering. In our evaluation, we verify these same policies with and without applicability filtering, to allow direct comparison of verification performance. As goal condition G for reach-avoid, we set the goal used by VEA during training. Training episodes stop at G, which exactly matches reach-avoid semantics.

Configurations. We compare a range of algorithmic configurations combining different enhancements for abstract transition computation with applicability filter and reach-avoid as part of VEA's verification algorithm.

- NoOpt disables and AllOpts enables all enhancements.
- OnlyPerOp, OnlySlack, OnlyOp, OnlyGen only enables PER-OP-DISJ, OPT-SLACK-VAR, ENTAIL-OP, ENTAIL-GEN respectively.
- NoPerOp, NoSlack, NoOp, NoGen enables all enhancements except PER-OP-DISJ, OPT-SLACK-VAR, ENTAIL-OP, ENTAIL-GEN respectively.
- Vea verifies the policy without applicability filtering and without reach-avoid as done by VEA.

With vs. without enhancements. Table 1 shows our results. AllOpts clearly dominates NoOpt. The latter only terminates on the smallest problem instances, with a runtime offset of up to three orders of magnitude.

Ablation study. OnlyPerOp covers 10 additional instances compared to NoOpt. In addition, OnlyPerOp always decreases runtime by at least one order of magnitude on instances covered by NoOpt. This indicates that the choice of encoding (PER-OP-DISJ or not) is a crucial factor for efficiency. That said, also the other configurations with a single enhancement, especially OnlyGen, increase coverage compared to NoOpt. OnlyGen and OnlyOp often decrease runtime by at least one order of magnitude. OnlySlack is less successful, usually performing on par with NoOpt. AllOpts outperforms every single-enhancement configuration and always covers additional instances. This shows that also the combination of enhancements is crucial.

NoPerOp performs competitive on 8-puzzle and smaller Blocksworld instances, but fails on larger ones similar to NoOpt. On Transport it performs consistently slower than AllOpts. Again, this demonstrates the relevance of PER-OP-DISJ. NoOp tends to be more efficient than NoGen. This indicates that ENTAIL-GEN is more crucial than ENTAIL-OP. NoSlack usually performs on par with AllOpts. In line with the results for OnlySlack, this shows that OPT-SLACK-VAR is the least crucial enhancement.

While AllOpts does never dominate any "all-but-one-enhancement" configuration over all instances, it always dominates in terms of accumulated runtime. This demonstrates that on average enabling all enhancements is more successful.

Comparison to Vea. Clearly, the additional complexity of applicability filtering and reach-avoid in SMT can increase verification time. On Blocksworld, AllOpts is worse than Vea, covering four instances less. On 8-puzzle, on the other hand, AllOpts covers three more instances than Vea and is competitive on the remaining ones. This is presumably due to the actual verification *results* – on NN 32 (cost-ign), the policy is safe without applicability filtering, but is unsafe with applicability filtering. This exemplifies that, without applicability filtering, a policy may be safe due to stalling. This questionable form of safety is no longer possible under applicability filtering. Presumably, the same issue occurs in the 8-puzzle instances not covered by Vea. On Blocksworld

Benchmark	NN	Safe	Time										Vea	
			NoOpt	OnlyPerOp	OnlySlack	OnlyOp	OnlyGen	NoPerOp	NoSlack	NoOp	NoGen	AllOpts	Safe	Time
	16	✓	8550	27	8654	24	18	19	**17**	18	18	**17**	✓	5
4 Blocks	32	✓	14568	87	14545	91	45	42	**30**	33	71	31	✓	8
(cost-ign)	64	✓	29202	1534	29153	1306	226	222	217	218	1294	**214**	✓	14
	16	✓	-	39090	-	-	-	-	23013	26605	23385	**22756**	✓	98
6 Blocks	32	✓	-	21132	-	-	-	-	**7556**	8043	11486	7620	✓	68
(cost-ign)	64	?	-	-	-	-	-	-	-	-	-	-	✓	613
	16	?	-	-	-	-	-	-	-	-	-	-	✓	7918
8 Blocks	32	?	-	-	-	-	-	-	-	-	-	-	?	-
(cost-ign)	64	?	-	-	-	-	-	-	-	-	-	-	?	-
	16	×	-	77	1242	-	216	79	70	**67**	70	**67**	×	38
8-puzzle	32	×	-	13820	-	-	14868	12086	**12022**	12810	12336	12263	✓	15789
(cost-ign)	64	×	-	12921	-	-	13519	**10752**	11800	11885	11686	11309	?	-
	16	✓	41459	113	42023	95	44	41	41	44	87	**40**	✓	27
4 Blocks	32	✓	-	4011	-	2690	**417**	421	429	432	2617	426	✓	311
(cost-awa)	64	?	-	-	-	-	-	-	-	-	-	-	✓	36369
	16	✓	-	-	-	-	-	-	23848	27013	33597	**23530**	✓	7374
6 Blocks	32	?	-	-	-	-	-	-	-	-	-	-	✓	25019
(cost-awa)	64	?	-	-	-	-	-	-	-	-	-	-	?	-
	16	×	-	964	-	-	-	-	671	**658**	684	674	×	82
8 Blocks	32	?	-	-	-	-	-	-	-	-	-	-	?	-
(cost-awa)	64	?	-	-	-	-	-	-	-	-	-	-	?	-
	16	×	-	2096	-	-	5092	1857	1742	1833	1769	**1738**	×	2411
8-puzzle	32	×	-	11716	-	-	12572	**10205**	10399	10976	10455	10226	?	-
(cost-awa)	64	×	-	35663	-	-	31834	**28836**	31430	30264	31667	28908	?	-
	16	×	24	0.5	24	24	10	10	0.5	**0.4**	0.5	0.5	×	0.3
Transport	32	×	25	**1**	25	25	11	11	**1**	**1**	**1**	**1**	×	0.5
	64	×	42	**0.5**	42	42	23	23	1	**0.5**	1	1	×	0.4

Table 1: Runtime results in seconds for the evaluated configurations of enhancements for applicability filtering with reach-avoid over different benchmarks and NN policies. (distinguishing cost-aware policies and cost-ignoring policies where applicable). - indicates runs that exceed the resource limit of 12h time and 4 GB memory. Vea shows results for verification without applicability-filtering and without reach-avoid.

and Transport, there are no such verification result differences. In particular, on the former many policies are safe with and without applicability filtering.

Unlike applicability filtering, reach-avoid does not affect the safety results for the verified policies. That is, any policy safe under reach-avoid is also safe for Vea. In other words, VEA's policies are safe "behind" the goal. Furthermore, we remark that verifying reach-avoid adds no significant runtime overhead compared to applicability filtering without reach-avoid, i.e., the increase in complexity compared to Vea is dominated by applicability filtering. We provide additional results (applicability filtering without reach-avoid and reach-avoid without applicability filtering) in the appendix.

7 Conclusion

The verification of neural action policies is important. Here we contribute enhancements for PPA with applicability filtering, getting rid of much of the additional complexity suffered by a baseline implementation. We also show how to verify safety for the more practical reach-avoid setting.

Important future directions for PPA include liveness properties, in particular the guarantee that a policy will eventually reach the goal; partial safety verification, continuing CEGAR on instances already proved to be unsafe, in order to identify safe regions of the state space; and the extension to probabilistic and/or continuous-state transition systems. Our enhancements are orthogonal to all these extensions.

Acknowledgments

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

References

Akintunde, M.; Lomuscio, A.; Maganti, L.; and Pirovano, E. 2018. Reachability Analysis for Neural Agent-Environment Systems. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona 30 October - 2 November 2018*, 184–193. AAAI Press.

Akintunde, M. E.; Kevorchian, A.; Lomuscio, A.; and Pirovano, E. 2019. Verification of RNN-Based Neural Agent-Environment Systems. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii USA, January 27 - February 1, 2019*, 6006–6013. AAAI Press.

Amir, G.; Schapira, M.; and Katz, G. 2021. Towards Scalable Verification of Deep Reinforcement Learning. In *For-*

mal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021, 193–203. IEEE.

Barrett, C. W.; and Tinelli, C. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*, 305–343. Springer.

Budde, C. E.; Dehnert, C.; Hahn, E. M.; Hartmanns, A.; Junges, S.; and Turrini, A. 2017. JANI: Quantitative Model and Tool Interaction. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *LNCS*, 151–168.

Cavada, R.; Cimatti, A.; Dorigatti, M.; Griggio, A.; Mariotti, A.; Micheli, A.; Mover, S.; Roveri, M.; and Tonetta, S. 2014. The nuXmv Symbolic Model Checker. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *LNCS*, 334–342. Springer.

Clarke, E.; Grumberg, O.; Jha, S.; Lu, Y.; and Veith, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *JACM*, 50(5): 752–794.

Dutta, S.; Chen, X.; and Sankaranarayanan, S. 2019. Reachability analysis for neural feedback systems using regressive polynomial rule inference. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*, 157–168. ACM.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15 2019*, 631–636. AAAI Press.

Graf, S.; and Saïdi, H. 1997. Construction of Abstract State Graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *LNCS*, 72–83. Springer.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*, 408–416. AAAI Press.

Huang, S.; Fan, J.; Li, W.; Chen, X.; and Zhu, Q. 2019. ReachNN: Reachability analysis of neural-network controlled systems. *ACM Trans. Embed. Comput. Syst.*, 18(5s): 106:1–106:22.

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*, 422–430. AAAI Press.

Ivanov, R.; Carpenter, T. J.; Weimer, J.; Alur, R.; Pappas, G. J.; and Lee, I. 2021. Verifying the Safety of Autonomous Systems with Neural Network Controllers. *ACM Trans. Embed. Comput. Syst.*, 20(1): 7:1–7:26.

Katz, G.; Barrett, C. W.; Dill, D. L.; Julian, K.; and Kochenderfer, M. J. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *LNCS*, 97–117. Springer.

Katz, G.; Huang, D. A.; Ibeling, D.; Julian, K.; Lazarus, C.; Lim, R.; Shah, P.; Thakoor, S.; Wu, H.; Zeljic, A.; Dill, D. L.; Kochenderfer, M.; and Barrett, C. 2019. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *LNCS*, 443–452. Springer.

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. *Nat.*, 518(7540): 529–533.

Singh, G.; Gehr, T.; Püschel, M.; and Vechev, M. T. 2019. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3(POPL): 41:1–41:30.

Stahlberg, S.; Bonet, B.; and Geffner, H. 2022. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. In *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling, ICAPS 2022, Singapore (virtual), June 13-24 2022*, 629–637. AAAI Press.

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

Tran, H.; Cai, F.; Lopez, D. M.; Musau, P.; Johnson, T. T.; and Koutsoukos, X. D. 2019. Safety Verification of Cyber-Physical Systems with Reinforcement Learning Control. *ACM Trans. Embed. Comput. Syst.*, 18(5s): 105:1–105:22.

Vinzent, M.; and Hoffmann, J. 2024. Neural Policy Safety Verification via Predicate Abstraction: Applicability Filtering. In *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2024*, Technical report and tool available at https://fai.cs.uni–saarland.de/vinzent/publications. AAAI Press.

Vinzent, M.; Sharma, S.; and Hoffmann, J. 2023. Neural Policy Safety Verification via Predicate Abstraction: CEGAR. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, 15188–15196. AAAI Press.

Vinzent, M.; Steinmetz, M.; and Hoffmann, J. 2022. Neural Network Action Policy Verification via Predicate Abstraction. In *Proceedings of the Thirty-Second International*

Conference on Automated Planning and Scheduling, ICAPS 2022, Singapore (virtual), June 13-24 2022. AAAI Press.

Vinzent, M.; Wu, M.; Wu, H.; and Hoffmann, J. 2023. Policy-Specific Abstraction Predicate Selection in Neural Policy Safety Verification. In *ICAPS Workshop on Reliable Data-Driven Planning and Scheduling (RDDPS) & The Verifying Learning AI Systems (VeriLearn) Workshop @ECAI*.

A Additional Experiments

Applicability Filtering without Reach-Avoid. Table 2 shows results for a complementary evaluation of our enhancements for applicability filtering without reach-avoid. All observations for verification of applicability filtering with reach-avoid (Table 1) directly translate to verification without reach-avoid (Table 2). In particular, the verification results are identical, i.e., VEA's policies are safe "behind" the goal. The runtime results are largely similar, i.e., reach-avoid adds no significant runtime overhead. There are instances where verification without reach-avoid takes longer. This is presumably due to internal heuristics of the SMT solver.

Reach-Avoid without Applicability Filtering. Table 3 shows results for reach-avoid without applicability filtering. In accordance with our previous observations, AllOpts performance on par with Vea. Safety results are identical.

AllOpts covers two additional instances compared to NoOpt. There is a consistent runtime speedup of up to one order of magnitude. This demonstrates that our enhancements are crucial for reach-avoid. That said, the efficiency gain is clearly more substantial for applicability filtering.

The ablation study shows that ENTAIL-GEN is most relevant: OnlyGen performs particularly well under the configurations with a single enhancement enabled, NoGen performs particularly bad under the configurations with a single enhancement disabled. Conversely, OnlySlack and OnlyOp perform on par with NoOpt, while NoSlack and NoOp perform on par with AllOpts, indicating that OPT-SLACK-VAR and ENTAIL-OP are negligible for reach-avoid. In fact, NoSlack tends to perform slightly better than AllOpts, suggesting that OPT-SLACK-VAR is actually counterproductive for reach-avoid without applicability filtering.

B Neural Action Policy

A (ReLU) feed-forward *neural network* for Θ is a (real-valued) function

$$f_\pi\colon \mathcal{S} \to \mathbb{R}^{d_d}, s \mapsto f_d(\dots f_2(f_1(s))),$$

where d denotes the number of layers in the NN, d_i for $i \in \{1,\dots,d\}$ denotes the size of layer i, and

- $f_1\colon \mathcal{S} \to \mathbb{R}^{d_1}, s \mapsto (s(v^1),\dots,s(v^{d_1}))$ is the *input layer* function, where $v^j \in \mathcal{V}$ for $j \in \{1,\dots,d_1\}$ denotes the state variable associated with input neuron j.
- $f_i\colon \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}, V \mapsto ReLU(W_i \cdot V + B_i)$, for $i \in \{2,\dots,d-1\}$, is the function of *hidden layer* i. W_i is the weight matrix of layer i, i.e., $(W_i)_{j,k}$ denotes the weight of the output of neuron k in layer $i-1$ to the input of neuron j in layer i. B_i is the bias vector, i.e., $(B_i)_j$ denotes the bias of neuron j in layer i.
- $f_d\colon \mathbb{R}^{d_{d-1}} \to \mathbb{R}^{d_d}, V \mapsto W_d \cdot V + B_d$ is the function of output layer d. Here, no ReLU activation is applied.

The *neural action policy* implemented by f_π is the function

$$\pi\colon \mathcal{S} \to \mathcal{L}, s \mapsto \begin{cases} \underset{\{l\in\mathcal{L} \mid \exists o\in\mathcal{O}_l\colon s\models o\}}{\operatorname{argmax}} \; f_\pi^l(s) & \text{if } \exists o \in \mathcal{O}\colon s \models o \\ \tau & \text{otherwise} \end{cases}$$

where f_π^l denotes the output of f_π associated with l (abbreviated π_l in the main text). The *noop*-label $\tau \in \mathcal{L}$ with $\mathcal{O}_\tau = \emptyset$ is selected iff s is terminal.[3] This in particular pertains to goal states in reach-avoid verification.

C Abstract Transition Problem in SMT

In this section, we outline the SMT encoding of the abstract transition problem, i.e., given operator $o = (l, g, u)$[4] does there exist a concrete state $s \in [s_\mathcal{P}]$ such that $s \models o$, $s[\![o]\!] \in [s'_\mathcal{P}]$ and $\pi(s) = l$. Importantly, our encoding differs from the encoding used by VEA only in the label selection of the policy and the non-goal condition for reach-avoid.

Each state variable $v \in \mathcal{V}$, occurs in an *unprimed* form; representing the state variable in the source state and a *primed* form v' representing the updated state variable in the successor state.

To encode the neural network structure we introduce *real-valued* auxiliaries variables:

$$\{v_{i,j} \mid i \in \{1,\dots,d\}, j \in \{1,\dots,d_i\}\}$$

and

$$\{v^{i,j} \mid i \in \{2,\dots,d-1\}, j \in \{1,\dots,d_i\}\}$$

corresponding to neuron inputs and outputs. More precisely, $v_{i,j}$ corresponds to the neuron output and $v^{i,j}$ to the input of hidden layer neurons. For $i = 1$, $v_{i,j}$ is syntactic sugar for the respective state variable v^j in the input layer.

The abstract transition problem is then encoded by the conjunction of the constraints:

(i) $lo_v \le v$ and $v \le up_v$ as well as $lo_v \le v'$ and $v' \le up_v$ for each $v \in \mathcal{V}$, where lo_v denotes the lower bound and up_v denotes the upper bound of state variable v.

(ii) p if $s_\mathcal{P}(p) = 1$ and $\neg p$ if $s_\mathcal{P}(p) = 0$ as well as p' if $s'_\mathcal{P}(p) = 1$ and $\neg p'$ if $s'_\mathcal{P}(p) = 0$ for each p in $\mathcal{P}$ where p' denotes the predicate in its primed form, i.e., with primed variables.

(iii) $\bigwedge_{i\in\{1,\dots,m\}} g_o^i$

(iv) $v' = u(v)$ for each $v \in \mathcal{V}$

(v) $v^{i,j} = \sum_{k=1}^{d_{i-1}} (W_i)_{j,k} \cdot v_{i-1,k} + (B_i)_j$ and $v_{i,j} = ReLU(v^{i,j})$ for each hidden layer $i \in \{2,\dots,d-1\}$ and each neuron $j \in \{1,\dots,d_i\}$,

[3] *Only if* since $\mathcal{O}_\tau = \emptyset$ and thus τ is never applicable.

[4] VEA apply SMT checks on a per-operator basis and iterate operators as part of their search algorithm, cf. Section E.

Benchmark	NN	Safe	Time										Vea	
			NoOpt	OnlyPerOp	OnlySlack	OnlyOp	OnlyGen	NoPerOp	NoSlack	NoOp	NoGen	AllOpts	Safe	Time
	16	✓	8577	25	8478	24	18	18	**17**	**17**	**17**	**17**	✓	5
4 Blocks	32	✓	16324	89	16118	94	45	46	35	36	74	**34**	✓	8
(cost-ign)	64	✓	27975	1481	28347	1233	227	222	214	**212**	1205	216	✓	14
	16	✓	-	38453	-	-	-	-	**22103**	25058	23049	22193	✓	98
6 Blocks	32	✓	-	22354	-	-	-	-	8251	8400	11445	**8093**	✓	68
(cost-ign)	64	?	-	-	-	-	-	-	-	-	-	-	✓	613
	16	?	-	-	-	-	-	-	-	-	-	-	✓	7918
8 Blocks	32	?	-	-	-	-	-	-	-	-	-	-	?	-
(cost-ign)	64	?	-	-	-	-	-	-	-	-	-	-	?	-
	16	×	-	78	1237	-	202	78	68	69	67	**65**	×	38
8-puzzle	32	×	-	13380	-	-	14488	**12270**	12352	12860	12620	12677	✓	15789
(cost-ign)	64	×	-	12812	-	-	13195	**10735**	11870	11379	11757	11137	?	-
	16	✓	41597	111	40589	88	41	41	42	42	79	**40**	✓	27
4 Blocks	32	✓	-	3958	-	2520	**416**	420	420	437	2529	417	✓	311
(cost-awa)	64	?	-	-	-	-	-	-	-	-	-	-	✓	36369
	16	✓	-	-	-	-	-	-	**23507**	25387	32917	23958	✓	7374
6 Blocks	32	?	-	-	-	-	-	-	-	-	-	-	✓	25019
(cost-awa)	64	?	-	-	-	-	-	-	-	-	-	-	?	-
	16	×	-	975	-	-	-	-	683	**647**	693	674	×	82
8 Blocks	32	?	-	-	-	-	-	-	-	-	-	-	?	-
(cost-awa)	64	?	-	-	-	-	-	-	-	-	-	-	?	-
	16	×	-	2073	-	-	4907	1843	1823	1799	1815	**1717**	×	2411
8-puzzle	32	×	-	10932	-	-	11868	**10063**	10574	10971	10533	10483	?	-
(cost-awa)	64	×	-	35034	-	-	32008	**28749**	29604	29769	31437	28800	?	-
	16	×	24	0.5	24	24	10	10	**0.4**	0.5	0.5	0.5	×	0.3
Transport	32	×	25	**1**	26	25	11	11	**1**	**1**	**1**	**1**	×	0.5
	64	×	42	**0.5**	42	42	23	22	1	**0.5**	**0.5**	**0.5**	×	0.4

Table 2: Runtime results in seconds for the evaluated configurations of enhancements for **applicability filtering without reach-avoid** over different benchmarks and NN policies. (distinguishing cost-aware policies and cost-ignoring policies where applicable). - indicates runs that exceed the resource limit of 12h time and 4 GB memory. Vea shows results for verification without applicability-filtering and without reach-avoid.

(vi) $v_{d,j} = \sum_{k=1}^{d_{d-1}} (W_d)_{j,k} \cdot v_{d-1,k} + (B_d)_j$ for the output layer d and each neuron $j \in \{1, \ldots, d_d\}$,

(vii) $\bigwedge_{l' \in \mathcal{L} \setminus \{l\}} \left(v_{d,j} > v_{d,k} \vee \neg \bigvee_{o \in \mathcal{O}_{l'}} \bigwedge_{i \in \{1,\ldots,m\}} g_o^i \right)$ where $j \in \{1, \ldots, d_d\}$ is the output neuron associated with l and $k \in \{1, \ldots, d_d\} \setminus \{j\}$ is the output neuron associated with l' (abbreviated π_l and $\pi_{l'}$ in the main text),

(viii) $\bigvee_i \neg G_i$.

(i) constrains the variables to respect the corresponding state variable domains, such that every satisfying assignment to the SMT encoding corresponds to a valid state pair s, s'. (ii) then encodes $s \in [s_\mathcal{P}]$ and $s' \in [s'_\mathcal{P}]$. (iii) encodes $s \models o$, and (iv) encodes $s' = s[\![o]\!]$. $\pi(s) = l$ is encoded by (v – vi, neural network) and (vii, label selection) – applicability of label l itself is entailed by $s \models o$ (iii). (viii) encodes the non-goal condition for reach-avoid.

Note that the presented encoding is specific to the NN-tailored solver *Marabou* (Katz et al. 2019) in that it assumes a special construct for ReLU constraints. Furthermore, *Marabou* only supports real-valued variables, i.e., integer state variables are continuously-relaxed. VEA establish integer support via a branch & bound loop around *Marabou* (Vinzent, Steinmetz, and Hoffmann 2022).

D Proofs

We attach a formal proof that slack variable transformation as per OPT-SLACK-VAR preserves satisfiability.

Proposition 1. Let $\mathrm{C} = \{c_1, \ldots, c_n\}$. For any assignment $s \in \mathcal{S}$ such that

$$s \models \bigwedge_{c \in \mathrm{C}} \sum_{v \in \mathcal{V}} d_v \cdot v \geq c$$

there exists an assignment $\hat{s}$ over $\mathcal{V} \cup \{a\}$ such that

$$\hat{s} \models \sum_{v \in \mathcal{V}} d_v \cdot v + a = 0$$

$$\hat{s} \models \bigwedge_{c \in \mathrm{C}} a \leq -c$$

and vice versa.

Proof. Let s over $\mathcal{V}$ such that $(\sum_{v \in \mathcal{V}} d_v \cdot v)(s) \geq c$ for each $c \in \mathrm{C}$. We set $\hat{s} = s \cup \{a \mapsto -(\sum_{v \in \mathcal{V}} d_v \cdot v)(s)\}$. It immediately follows $(\sum_{v \in \mathcal{V}} d_v \cdot v + a)(\hat{s}) = 0$. Moreover, for each $c \in \mathrm{C}$,

Benchmark	NN	Safe	Time								Vea	
			NoOpt	OnlySlack	OnlyOp	OnlyGen	NoSlack	NoOp	NoGen	AllOpts	Safe	Time
	16	$\checkmark$	17	17	17	**7**	**7**	**7**	17	**7**	$\checkmark$	5
4 Blocks	32	$\checkmark$	36	36	36	**8**	**8**	**8**	36	9	$\checkmark$	8
(cost-ign)	64	$\checkmark$	206	207	201	15	**14**	15	202	**14**	$\checkmark$	14
	16	$\checkmark$	151	151	148	102	**99**	101	147	109	$\checkmark$	98
6 Blocks	32	$\checkmark$	265	266	263	71	**70**	**70**	263	**70**	$\checkmark$	68
(cost-ign)	64	$\checkmark$	8745	8781	8562	623	**615**	626	8616	622	$\checkmark$	613
	16	$\checkmark$	11072	11621	10818	8189	**7767**	8011	10825	7961	$\checkmark$	7918
8 Blocks	32	?	-	-	-	-	-	-	-	-	?	-
(cost-ign)	64	?	-	-	-	-	-	-	-	-	?	-
	16	$\times$	45	45	47	**38**	**38**	39	45	**38**	$\times$	38
8-puzzle	32	$\checkmark$	40572	40694	41198	16498	15829	16304	39902	**15563**	$\checkmark$	15789
(cost-ign)	64	?	-	-	-	-	-	-	-	-	?	-
	16	$\checkmark$	77	77	77	**28**	**28**	**28**	77	29	$\checkmark$	27
4 Blocks	32	$\checkmark$	2566	2562	2553	316	**315**	319	2548	317	$\checkmark$	311
(cost-awa)	64	$\checkmark$	-	-	-	36455	**36396**	36438	-	36495	$\checkmark$	36369
	16	$\checkmark$	10208	10189	10060	7411	**7329**	7429	10104	7511	$\checkmark$	7374
6 Blocks	32	$\checkmark$	-	-	-	25496	**25096**	25470	-	25294	$\checkmark$	25019
(cost-awa)	64	?	-	-	-	-	-	-	-	-	?	-
	16	$\times$	404	404	402	83	**82**	**82**	402	83	$\times$	82
8 Blocks	32	?	-	-	-	-	-	-	-	-	?	-
(cost-awa)	64	?	-	-	-	-	-	-	-	-	?	-
	16	$\times$	4359	4478	4300	2486	**2410**	2451	4257	2418	$\times$	2411
8-puzzle	32	?	-	-	-	-	-	-	-	-	?	-
(cost-awa)	64	?	-	-	-	-	-	-	-	-	?	-
	16	$\times$	**0.4**	**0.4**	**0.4**	**0.4**	**0.4**	**0.4**	**0.4**	**0.4**	$\times$	0.3
Transport	32	$\times$	**0.4**	**0.4**	**0.4**	0.5	0.5	**0.4**	**0.4**	0.5	$\times$	0.5
	64	$\times$	3	3	3	**0.4**	**0.4**	**0.4**	3	**0.4**	$\times$	0.4

Table 3: Runtime results in seconds for the evaluated configurations of enhancements for **reach-avoid without applicability-filtering** over different benchmarks and NN policies. (distinguishing cost-aware policies and cost-ignoring policies where applicable). - indicates runs that exceed the resource limit of 12h time and 4 GB memory. Vea shows results for verification without applicability-filtering and without reach-avoid.

since $\sum_{v\in\mathcal{V}} d_v \cdot v \geq c \equiv -\sum_{v\in\mathcal{V}} d_v \cdot v \leq -c$, it also holds $\hat{s}(a) \leq -c$.

Let $\hat{s}$ over $\mathcal{V} \cup \{a\}$ such that $(\sum_{v\in\mathcal{V}} d_v \cdot v + a)(\hat{s}) = 0$, and $\hat{s}(a) \leq -c$ for each $c \in \mathrm{C}$. We set $s = \{v \mapsto \hat{s}(v) \mid v \in \mathcal{V}\}$. Since $\hat{s}(a) = -(\sum_{v\in\mathcal{V}} d_v \cdot v)(\hat{s}) = -(\sum_{v\in\mathcal{V}} d_v \cdot v)(s)$ it follows $-\hat{s}(a) = (\sum_{v\in\mathcal{V}} d_v \cdot v)(s)$. Moreover, for each $c \in \mathrm{C}$, since $a \leq -c \equiv -a \geq c$, it follows $(\sum_{v\in\mathcal{V}} d_v \cdot v)(s) \geq c$.

□

E Enhancements during Abstract State Expansion

Algorithm 1 shows an adapted version of VEA's abstract state expansion algorithm. Given an abstract state $s_{\mathcal{P}}$ to be expanded, VEA construct the SMT encoding ψ incrementally, iterating labels (line 7), operators (line 10) and successor candidates (line 14). We exploit the incremental nature of this process to apply our enhancements efficiently.

We test applicability for each operator o at expansion start (line 3) – originally VEA simply apply this test on-the-fly (line 11). In reach-avoid, we also test for satisfiability of G (line 2). We consider this an adapted version of ENTAIL-OP. If this test fails, $\neg G$ is entailed. Otherwise, we apply ENTAIL-GEN to enhance the encoding of $\neg G$.

We use the operator applicability information, to enhance the encoding of the policy selection condition (line 9), applying ENTAIL-OP, but also the other enhancements (PER-OP-DISJ, OPT-SLACK-VAR, ENTAIL-GEN). Here, incrementality enables us to reuse one round of enhancements over multiple transition tests (line 18). During the transition test, we again apply the generic enhancements (OPT-SLACK-VAR, ENTAIL-GEN) – on the implementation level, as part of *Marabou*– exploiting potentially tightened variable bounds.

F CEGAR

To find a suitable predicate set $\mathcal{P}$ automatically, VEA provide a CEGAR framework tailored to PPA (Vinzent, Sharma, and Hoffmann 2023). Starting from an initially coarse predicate set $\mathcal{P} = \emptyset$, they iteratively search for an abstract unsafe path $\sigma_{\mathcal{P}}$ in $\Theta^{\pi}_{\mathcal{P}}$. If $\sigma_{\mathcal{P}}$ is spurious, i.e., without concretization in Θ under π, short Θ^{π}, they refine $\mathcal{P}$ by adding predicates based on the source of spuriousness in $\sigma_{\mathcal{P}}$, and iterate. If $\sigma_{\mathcal{P}}$ is not spurious then π is proven unsafe. Conversely, if no $\sigma_{\mathcal{P}}$ exists then π is proven safe.

VEA introduce a new technique for refinement of policy-induced spuriousness that adds predicates based on a *concretization* state $s_c \in \mathcal{S}$ reachable in Θ under π and an

Algorithm 1: Abstract state expansion (Vinzent, Steinmetz, and Hoffmann 2022) – adapted version and illustration.

Input: $s_{\mathcal{P}} \in \mathcal{S}_{\mathcal{P}}$

$\psi \leftarrow s \in [s_{\mathcal{P}}]$ `// Incremental SMT encoding.`
if $\mathit{check}(\psi \wedge s \models G)$ **then** $\psi \leftarrow \psi \wedge \mathit{enhance}(\neg G)$
for *each* $o \in \mathcal{O}$ **do**
    if $\mathit{check}(\psi \wedge s \models o)$ **then** mark o applicable
    else mark o inapplicable
end
for *each* $l \in \mathcal{L}$ **do**
    $\mathit{push}(\psi)$ `// Incremental stack.`
    $\psi \leftarrow \psi \wedge \mathit{enhance}(\pi(s) = l)$
    for *each* $o \in \mathcal{O}$ *with* $o = (l, g, u)$ **do**
        if *o is marked inapplicable* **then continue**
        $\mathit{push}(\psi)$
        $\psi \leftarrow \psi \wedge s \models o$
        for *each successor candidate* $s'_{\mathcal{P}} \in \mathcal{S}_{\mathcal{P}}$ **do**
            if $\mathit{reached}(s'_{\mathcal{P}})$ **then continue**
            $\mathit{push}(\psi)$
            $\psi \leftarrow \psi \wedge s[\![o]\!] \in [s'_{\mathcal{P}}]$
            if $\mathit{check}(\mathit{enhance}(\psi))$ **then**
                if $\mathit{check}(s' \in [s'_{\mathcal{P}}] \wedge s' \models \neg G \wedge \phi_u)$ **then**
                    trigger CEGAR
                end
                mark $s'_{\mathcal{P}}$ for expansion
            end
            $\mathit{pop}(\psi)$
        end
        $\mathit{pop}(\psi)$
    end
    $\mathit{pop}(\psi)$
end

(unreachable) abstract transition *witness* $s_w \in \mathcal{S}$. Follow-up research (Vinzent et al. 2023) finds that this refinement approach is prone to significant runtime variances. Specifically, s_w and s_c are extracted as solutions to SMT formulae, and are, hence, subject to internal heuristics of the deployed SMT solver – in general more than one solution exists. Different s_w, s_c may result in different $\mathcal{P}$ and, thereby, potentially varying runtime performance, which may accumulate over several CEGAR iterations.

In our ablation study, we experiment with different encoding optimizations, and, hence, potentially different SMT solutions. To enable comparability in our ablation study, we deploy an adapted version of VEAs CEGAR framework tailored to prevent *SMT-solution-induced* runtime variance. We find that the performance of this version is competitive with the original. Completeness is preserved. Algorithm 2 shows the adapted abstraction refinement. There are two key modifications.

1) In the original version, VEA check policy-induced spuriousness with respect to a specific concretization in Θ sampled as a SMT solution when checking for spuriousness induced by the transition semantics of Θ (line 2). While the overall CEGAR framework is complete, this check is incomplete – another concretization in Θ without policy-induced spuriousness may exist – and SMT-solution-dependent. The adapted version deploys a complete check for policy-induced spuriousness (line 8). Analogously to Θ-spuriousness, it incrementally checks for a prefix concretization i under π. On the implementation level, this check is encoded in SMT using the NN-tailored SMT solver *Marabou* (Katz et al. 2017). This *multi-π-step* encoding is feasible, since the policy selection $\langle o^0, \ldots, o^{i-1} \rangle$ is fixed. This is inherently different to SMT encodings for bounded-length verification (cf. (Vinzent, Steinmetz, and Hoffmann 2022)). In addition, multi-π-step encodings are enhanced by PER-OP-DISJ, OPT-SLACK-VAR and ENTAIL-GEN.

2) If concretization fails for some i, the detected counterexample is spurious. Here, lines 11 through 21 replace the witness-based refinement of the original version. We first compute state variable values that are entailed by the path semantics (lines 12 & 30), and – analogously to the original version – add predicates using weakest precondition computation (line 14). If new predicates are added (line 16), refinement concludes. If not, we add new predicates using binary search on the state variable domain (lines 18 & 38).

An additional modification under reach-avoid involves that existence checks in Algorithm 2 constrain $s^j \not\models \neg G$ at each path step $j \in \{0, \ldots, i\}$. Recall, formally $\neg G$ is encoded into g and ϕ_u, and, thereby, implicitly subsumed by $\langle s^0, o^0, \ldots, o^{i-1}, s^i \rangle \in \Theta$ and $s^i \models g^i$ respectively. On the implementation level, we check reach-avoid-induced spuriousness individually, i.e., via an additional existence check $\exists s^0, \ldots, s^i \in \mathcal{S} \colon s^0 \in [s^0_{\mathcal{P}}] \wedge s^0 \models \phi_0 \wedge \langle s^0, o^0, \ldots, o^{i-1}, s^i \rangle \in \Theta \wedge s^i \not\models \neg G$ (preceding line 2), and refine analogously to g^i-induced spuriousness (line 3), i.e., $\mathcal{P} \leftarrow \mathcal{P} \cup \mathtt{WP}(G,\ \langle o^0, \ldots, o^{i-1} \rangle)$.

Algorithm 2: Abstraction refinement (Vinzent, Sharma, and Hoffmann 2023) – adapted version.

Input: $s^0_{\mathcal{P}} \models \phi_0$, $\langle o^0, \ldots, o^{n-1} \rangle$ with $o^i = (g^i, l^i, u^i)$, and $g^n := \phi_u$.

// Check Θ-spuriousness.
for $i \in \{0, \ldots, n\}$ **do**
  if $\neg \exists s^0, \ldots, s^i \in \mathcal{S}\colon s^0 \in [s^0_{\mathcal{P}}] \wedge s^0 \models \phi_0 \wedge \langle s^0, o^0, \ldots, o^{i-1}, s^i \rangle \in \Theta \wedge s^i \models g^i$ **then**
    $\mathcal{P} \leftarrow \mathcal{P} \cup$ `WP` $(g^i,\ \langle o^0, \ldots, o^{i-1} \rangle)$
    return *SPURIOUS*
  end
end
// Check π-spuriousness.
for $i \in \{0, \ldots, n\}$ **do**
  if $\exists s^0, \ldots, s^n \in \mathcal{S}\colon s^0 \in [s^0_{\mathcal{P}}] \wedge s^0 \models \phi_0 \wedge \langle s^0, o^0, \ldots, o^{n-1}, s^n \rangle \in \Theta \wedge s^n \models \phi_u \wedge \langle s^0, o^0, \ldots, o^i, s^{i+1} \rangle \in \Theta^\pi$ **then**
    continue
  end
  $\mathcal{P}' \leftarrow \mathcal{P}$
  $\textit{entails} \leftarrow$ `CompEntail` $(s^0_{\mathcal{P}},\ \langle o^0, \ldots, o^{n-1} \rangle,\ i)$
  for $(v, c) \in \textit{entails}$ **do**
    $\mathcal{P} \leftarrow \mathcal{P} \cup$ `WP` $((v = c),\ \langle o^0, \ldots, o^{i-1} \rangle)$
  end
  if $\mathcal{P} \neq \mathcal{P}'$ **then return** *SPURIOUS*
  for $v \in \mathcal{V} \setminus \mathit{dom}(\textit{entails})$ **do**
    $p \leftarrow$ `ProposeSplit` $(v,\ lo_v,\ up_v)$
    if $p \neq$ *NONE* **then** $\mathcal{P} \leftarrow \mathcal{P} \cup \{p\}$
  end
  return *SPURIOUS*
end
return *NON-SPURIOUS*

Procedure `WP` $(\phi, \langle o^0, \ldots, o^{i-1} \rangle)$**:**
  $\phi^i_{wp} \leftarrow \phi$
  for $j \in \{i-1, \ldots, 0\}$ **do**
    $\phi^j_{wp} \leftarrow wp_{u^j}(\phi^{j+1}_{wp})$
  end
  return $\{\phi^0_{wp}, \ldots, \phi^i_{wp}\}$

Procedure `CompEntail` $(s^0_{\mathcal{P}}, \langle o^0, \ldots, o^{n-1} \rangle, i)$**:**
  $\textit{entails} \leftarrow \emptyset$
  let $s^0_c, \ldots, s^n_c \in \mathcal{S}\colon s^0_c \in [s^0_{\mathcal{P}}] \wedge s^0 \models \phi_0 \wedge \langle s^0_c, o^0, \ldots, o^{n-1}, s^n_c \rangle \in \Theta \wedge s^n \models \phi_u$
  for $v \in \mathcal{V}$ **do**
    if $\exists s^0, \ldots, s^n \in \mathcal{S}\colon s^0 \in [s^0_{\mathcal{P}}] \wedge s^0 \models \phi_0 \wedge \langle s^0, o^0, \ldots, o^{n-1}, s^n \rangle \in \Theta \wedge s^i(v) \neq s^i_c(v)$ **then continue**
    $\textit{entails} \leftarrow \textit{entails} \cup \{v \mapsto s^i_c(v)\}$
  end
  return $\textit{entails}$

Procedure `ProposeSplit` (v, l, u)**:**
  $I \leftarrow [(l, u)]$
  while $I \neq \emptyset$ **do**
    $(l, u) \leftarrow I.\textit{pop\_front}()$
    $c \leftarrow \lfloor (l+u)/2 \rfloor$
    if $(v \geq c) \notin \mathcal{P}$ **then return** $v \geq c$
    if $l < c$ **then** $I.\textit{push\_back}((l, c-1))$
    if $u > c$ **then** $I.\textit{push\_back}((c+1, u))$
  end
  return *NONE*