Neural Policy Safety Verification via Predicate Abstraction: CEGAR Technical Report

Marcel Vinzent,[1] **Siddhant Sharma,**[2] **Jörg Hoffmann**[1,3]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2] Department of Electrical Engineering, Indian Institute of Technology Delhi, New Delhi, India
[3] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
{vinzent, hoffmann}@cs.uni-saarland.de, Siddhant.Sharma.ee119@ee.iitd.ac.in

Abstract

Neural networks (NN) are an increasingly important representation of action policies π. Recent work has extended predicate abstraction to prove safety of such π, through *policy predicate abstraction (PPA)* which over-approximates the state space subgraph induced by π. The advantage of PPA is that reasoning about the NN – calls to SMT solvers – is required only locally, at individual abstract state transitions, in contrast to bounded model checking (BMC) where SMT must reason globally about sequences of NN decisions. Indeed, it has been shown that PPA can outperform a simple BMC implementation. However, the abstractions underlying these results (i.e., the abstraction predicates) were supplied manually. Here we automate this step. We extend *counterexample-guided abstraction refinement (CEGAR)* to PPA. This involves dealing with a new source of spuriousness in abstract unsafe paths, pertaining not to transition behavior but to the decisions of the neural network π. We introduce two methods tackling this issue based on the states involved, and we show that global SMT calls deciding spuriousness exactly can be avoided. We devise algorithmic enhancements leveraging incremental computation and heuristic search. We show empirically that the resulting verification tool has significant advantages over an encoding into the state-of-the-art model checker nuXmv. In particular, ours is the only approach in our experiments that succeeds in proving policies safe.

1 Introduction

Neural networks (NN) are an increasingly important representation of action policies, in particular in planning (Issakkimuthu, Fern, and Tadepalli 2018; Groshev et al. 2018; Garg, Bajpai, and Mausam 2019; Toyer et al. 2020). But how to verify that such a policy is safe? Given a policy π, a **start condition** ϕ_0, and an **unsafety condition** ϕ_u, how to verify whether a state $s^u \models \phi_u$ is reachable from a state $s^0 \models \phi_0$ under π? Such verification is potentially very hard as it compounds the state space explosion with the difficulty of analyzing even single NN decision episodes.

Research on this question still is in its early stages. A prominent line of works addresses neural controllers of dynamical systems, where the NN outputs a vector u of reals forming input to a continuous state-evolution function f (Sun, Khedr, and Shoukry 2019; Tran et al. 2019; Huang et al. 2019; Dutta, Chen, and Sankaranarayanan 2019). Recent work extends this thread to hybrid systems, addressing smooth (tanh/sigmoid) activation functions by compilation into such systems (Ivanov et al. 2021). Another thread explores bounded-length verification of neural controllers (Akintunde et al. 2018, 2019; Amir, Schapira, and Katz 2021).

Here we follow up on the work by Vinzent et al. (2022) (henceforth: **Vea22**), which tackles NN policies π with ReLU activation functions taking discrete action choices in non-deterministic state spaces over bounded-integer state variables. The approach extends **predicate abstraction (PA)** (Graf and Saïdi 1997; Ball et al. 2001; Henzinger et al. 2004) to what Vea22 baptise **policy predicate abstraction (PPA)**. Like PA, PPA builds an over-approximating abstraction defined through a set $\mathcal{P}$ of **predicates**, where each $p \in \mathcal{P}$ is a linear constraint over the state variables (e.g. $x = 7$ or $x \leq y$). Unlike PA however, PPA abstracts not the full state space Θ, but the **policy-restricted** state space Θ^π, i.e., the state-space subgraph containing only the transitions taken by π. Vea22's method builds the fragment of $\Theta^\pi_\mathcal{P}$ – the predicate abstraction of Θ^π – reachable from ϕ_0, and checks whether ϕ_u is reached. If this is not the case, then π is safe.[1]

Compared to **bounded model checking (BMC)**, which iteratively checks length-L safety through an encoding into SMT (de Moura and Bjørner 2008), the advantage of PPA is that the required SMT calls are much cheaper. PPA uses SMT to decide about the existence of individual abstract state transitions. While many such SMT calls are needed, each is *local* to a single NN decision, requiring a singly copy of the NN π in the SMT encoding. In contrast, BMC requires *global* SMT encodings pertaining to sequences of NN decision steps, with one copy of π in each step, which incurs a blow-up prohibitive in L. Indeed, Vea22 show that PPA can outperform a simple implementation of BMC.

The central weakness of this result, thus far, is that Vea22 supplied the underlying predicate sets $\mathcal{P}$ manually. They examined performance as a function of $|\mathcal{P}|$, with predicate sets scaled according to simple manually designed schemes. Their positive results pertain to particular points in these scaling schemes: particular instances of $\mathcal{P}$ that work well,

[1] Note here that this safety result is specific to ϕ_0, i.e., checking different start conditions requires to re-run PPA. The advantage is that, given a fixed ϕ_0, we can leverage restricted reachability under π. Vea22 show that this is crucial for PPA to be practicable.

which are typically located at sweetspots in the middle of the scaling scheme, with PPA being very costly, often infeasible, outside the sweetspot. *Can suitable $\mathcal{P}$ be found automatically, without incurring prohibitive overhead from computing abstract state spaces outside the sweetspot?*

Here we answer this question in the affirmative, through extending the well-known **counterexample-guided abstraction refinement (CEGAR)** (Clarke et al. 2003) paradigm to the PPA setting. The key challenge here is a new source of spuriousness in counterexamples. An abstract unsafe path may not have a concrete correspondence because, at some point along the path, *π's decision on the current concrete state s does not match the next abstract state transition*. We introduce two alternative methods tackling this issue, by introducing predicates allowing to distinguish s from (a) *all* other states (**concretization exclusion**) or (b) *a witness state s_w* where π does decide as required (**witness splitting**). We furthermore show that global SMT calls, which would be needed to decide exactly whether an abstract unsafe path has a concrete correspondence, can be avoided: as we show, even when checking only a single concretization candidate in each CEGAR iteration, the overall algorithm remains complete. We finally devise two algorithmic enhancements, leveraging incremental computation and heuristic search to improve individual CEGAR iterations.

We evaluate the resulting verification tool on Vea22's benchmarks, comparing against a broad range of state-of-the-art verification methods implemented in NUXMV (Cavada et al. 2014). NUXMV does not natively support NNs, but the neural policy π can be encoded into constraints in its input language. NUXMV is sometimes more effective than our tool in finding unsafe paths. But the opposite also happens, and, more importantly, *ours is the only approach in our experiments that succeeds in proving policies safe.* Our algorithmic enhancements help significantly to achieve this success, in particular heuristic search which often improves performance by orders of magitude.

2 Preliminaries

We consider the policy verification setting as introduced by Vea22. A state space is a tuple $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$ of **state variables** $\mathcal{V}$, **action labels** $\mathcal{L}$, and **operators** $\mathcal{O}$. For each variable $v \in \mathcal{V}$ the domain $D_v \neq \emptyset$ is a bounded integer interval. We denote by Exp the set of **linear integer expressions** over $\mathcal{V}$, of the form $d_1 \cdot v_1 + \cdots + d_r \cdot v_r + c$ with $d_1, \ldots, d_r, c \in \mathbb{Z}$. C denotes the set of **linear integer constraints** over $\mathcal{V}$, i.e., constraints of the form $e_1 \bowtie e_2$ with $\bowtie \in \{\leq, =, \geq\}$ and $e_1, e_2 \in \mathit{Exp}$, and Boolean combinations thereof (we use $e_1 < e_2$ as syntactic sugar for $\neg(e_1 \geq e_2)$, etc). An **operator** $o \in \mathcal{O}$ is a tuple (g, l, u) with **label** $l \in \mathcal{L}$, **guard** $g \in C$, and (partial) **update** $u \colon \mathcal{V} \to \mathit{Exp}$.

A (partial) **variable assignment** s over $\mathcal{V}$ is a function with domain $dom(s) \subseteq \mathcal{V}$ and $s(v) \in D_v$ for $v \in dom(s)$. Given s_1, s_2, we denote by $s_1[s_2]$ the update of s_1 by s_2, i.e., $dom(s_1[s_2]) = dom(s_1) \cup dom(s_2)$ with $s_1[s_2](v) = s_2(v)$ if $v \in dom(s_2)$, else $s_1[s_2](v) = s_1(v)$. By $e(s)$ we denote the evaluation of $e \in \mathit{Exp}$ over s, and by $\phi(s)$ the evaluation of $\phi \in C$. If $\phi(s)$ evaluates to true, we write $s \models \phi$.

The **state space** of $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$ is a labeled transition system (LTS) $\Theta = \langle \mathcal{S}, \mathcal{L}, \mathcal{T} \rangle$. 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 $o = (g, l, u)$ such that $s \models g$ and $s' = s[u(s)]$, where $u(s) = \{v \mapsto u(v)(s) \mid v \in dom(u)\}$. We also write $s \models o$ for $s \models g$, and abbreviate $s[\![o]\!]$ for $s[u(s)]$.

Observe that the separation between action labels and operators allows both, state-dependent effects (different operators with the same label l applicable in different states) as well as action outcome non-determinism (different operators with the same label l applicable in the same state).

A **policy** π is a function $\mathcal{S} \to \mathcal{L}$. We consider π represented by a **neural network (NN)**. While Vea22's approach applies, in principle, to arbitrary NN, its current implementation is restricted to feed-forward NN with **rectified linear unit (ReLU)** activation functions $ReLU(x) = \max(x, 0)$. These NN consist of an input layer, arbitrarily many hidden layers, and an output layer with one neuron per label l; π is obtained by applying argmax to the output layer.

We next introduce basic safety notations pertaining to Θ as a whole; we will later modify these to consider policy safety instead. A **safety property** is a pair $\rho = (\phi_0, \phi_u)$ where $\phi_0, \phi_u \in C$. Here, ϕ_u identifies the set of unsafe states that should be unreachable from the set of possible start states represented by ϕ_0. Θ is **unsafe** with respect to ρ iff there exists a path $\langle s^0, o^0, \ldots, s^{n-1}, o^{n-1}, s^n \rangle$ in Θ such that $s^0 \models \phi_0$ and $s^n \models \phi_u$. Otherwise Θ is **safe**.

Predicate abstraction (Graf and Saïdi 1997) is a well-established technique to prove safety. Assume a set of predicates $\mathcal{P} \subseteq C$. An **abstract state** $s_\mathcal{P}$ is a (complete) truth value assignment over $\mathcal{P}$. The abstraction of a (concrete) state $s \in \mathcal{S}$ is the abstract state $s|_\mathcal{P}$ with $s|_\mathcal{P}(p) = p(s)$ for each $p \in \mathcal{P}$. Conversely, $[s_\mathcal{P}] = \{s' \in \mathcal{S} \mid s'|_\mathcal{P} = s_\mathcal{P}\}$ denotes the **concretization** of abstract state $s_\mathcal{P}$, i.e., the set of all concrete state represented by $s_\mathcal{P}$. The **predicate abstraction** of Θ over $\mathcal{P}$ is then the LTS $\Theta_\mathcal{P} = \langle \mathcal{S}_\mathcal{P}, \mathcal{L}, \mathcal{T}_\mathcal{P} \rangle$, where $\mathcal{S}_\mathcal{P}$ is the set of all abstract states over $\mathcal{P}$, and $\mathcal{T}_\mathcal{P} = \{(s|_\mathcal{P}, l, s'|_\mathcal{P}) \mid (s, l, s') \in \mathcal{T}\}$. To compute $\Theta_\mathcal{P}$, one must solve what we call the **abstract transition problem** for every possible abstract transition: $(s_\mathcal{P}, l, s'_\mathcal{P}) \in \mathcal{T}_\mathcal{P}$ iff there exists an operator $o \in \mathcal{O}$ with label l, and a concrete state $s \in [s_\mathcal{P}]$, such that $s \models o$ and $s[\![o]\!] \in [s'_\mathcal{P}]$. Such abstract transition problems are routinely encoded into satisfiability modulo theories (SMT) (Barrett and Tinelli 2018), using off-the-shelf solvers like $Z3$ (de Moura and Bjørner 2008).

Analogously to safety in Θ, the abstract state space $\Theta_\mathcal{P}$ is said to be **unsafe** with respect to a safety property $\rho = (\phi_0, \phi_u)$ iff there exists a path $\langle s^0_\mathcal{P}, o^0, \ldots, o^{n-1}, s^n_\mathcal{P} \rangle$ in $\Theta_\mathcal{P}$ such that $s^0_\mathcal{P} \models \phi_0$ and $s^n_\mathcal{P} \models \phi_u$, where $s_\mathcal{P} \models \phi$ iff there exists $s \in [s_\mathcal{P}]$ such that $s \models \phi$. Otherwise $\Theta_\mathcal{P}$ is **safe**, in which case Θ must be safe as well.

An abstract unsafe path in $\Theta_\mathcal{P}$ may be **spurious**, i.e., without concretization in Θ. Counterexample-guided abstraction refinement (**CEGAR**) (e.g. (Clarke et al. 2003; Henzinger et al. 2004; Gupta and Strichman 2005; Podelski and Rybalchenko 2007)) 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.

3 Vea22's Methods and Results

We next briefly revisit Vea22 as the direct background for our work. Safety of a given policy π is proved via safety of the **policy-restricted state space** Θ^π, the subgraph $\langle \mathcal{S}, \mathcal{L}, \mathcal{T}^\pi \rangle$ of Θ with $\mathcal{T}^\pi = \{(s,l,s') \in \mathcal{T} \mid \pi(s) = l\}$. In turn, safety of Θ^π is proved via safety of the **policy predicate abstraction** of Θ^π, $\Theta^\pi_\mathcal{P} = \langle \mathcal{S}_\mathcal{P}, \mathcal{L}, \mathcal{T}^\pi_\mathcal{P} \rangle$ where $\mathcal{T}^\pi_\mathcal{P} = \{(s|_\mathcal{P}, l, s'|_\mathcal{P}) \mid (s,l,s') \in \mathcal{T}, \pi(s) = l\}$.

Vea22 introduce an algorithm to compute the fragment of $\Theta^\pi_\mathcal{P}$ reachable from the start-state constraint ϕ_0. The abstract transition problems in this computation involve checking whether π selects the correct label: $(s_\mathcal{P}, l, s'_\mathcal{P}) \in \mathcal{T}_\mathcal{P}$ iff there exists an operator $o \in \mathcal{O}$ with label l, and $s \in [s_\mathcal{P}]$, such that $s \models o$ and $s[\![o]\!] \in [s'_\mathcal{P}]$ *and* $\pi(s) = l$. This is a key new source of complexity as the SMT sub-formula representing the NN π contains one disjunction for every neuron. Vea22 show how this can be dealt with through approximate SMT checks. In particular, this pertains to continuous relaxations of the integer state variables, solved with *Marabou* (Katz et al. 2019), an SMT solver tailored to NN analysis.

Vea22 do not provide a method to automatically find the predicate set $\mathcal{P}$. Instead, for their empirical evaluation, they use manually designed predicate sets $\mathcal{P}$, consisting of box constraints $v \geq c$. They scale these $\mathcal{P}$ according to simple schemes, and examine performance as a function of $|\mathcal{P}|$. Figure 1 gives an illustrative excerpt of their results.

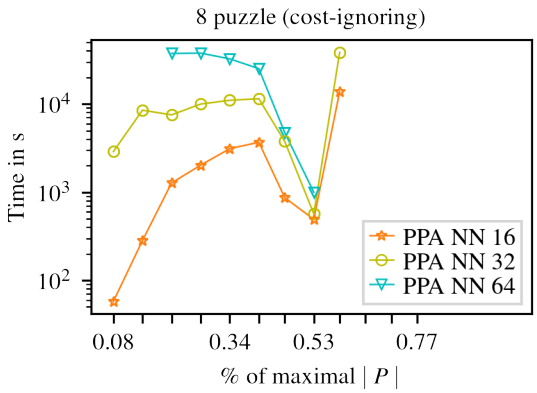

Figure 1: Excerpt of results by Vea22, on one of their benchmarks, for NN policies of different hidden-layer sizes. PPA is their best-performing policy predicate abstraction variant.

The runtime to compute the reachable fragment of $\Theta^\pi_\mathcal{P}$ clearly has a sweetspot in the middle of the $|\mathcal{P}|$ range. This is highly characteristic. To the left of the sweetspot, $\mathcal{P}$ is coarse so the NN input regions are large which results in expensive SMT calls (especially for larger NN). This effect is reduced as $\mathcal{P}$ becomes finer; in addition, with the abstraction becoming more fine-grained, the reduced reachability under π kicks in. Eventually however, the state-space explosion in $|\mathcal{P}|$ outweighs these benefits, resulting in exponential runtime growth to the right of the sweetspot.

In summary, these results by Vea22 are encouraging, yet leave us with the question whether *suitable $\mathcal{P}$ can be found automatically, without incurring prohibitive overhead from computing abstract state spaces outside the sweetspot*. We now show how to answer this question in the affirmative.

4 Policy CEGAR: Spuriousness

We contribute a CEGAR method for policy predicate abstraction. Next we discuss, at an intuitive level, the new source of spuriousness relative to standard predicate abstraction, and how we deal with it. In Section 5 we specify our method formally and prove its completeness.

4.1 Sources of Spuriousness

In standard predicate abstraction, all spuriousness is due to over-approximated transition behavior. In policy predicate abstraction however, a path may be spurious due to policy decisions even if the required transition behavior is present. Figure 2 illustrates this.

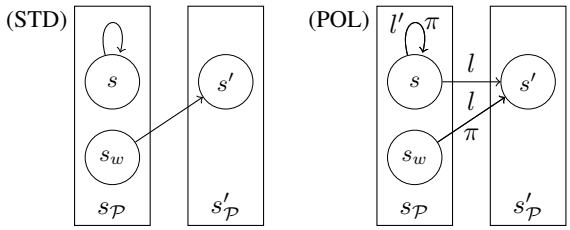

Figure 2: Illustration of standard spuriousness (STD) and policy-induced spuriousness (POL). $s \in [s_\mathcal{P}]$ is the concrete state reached by a candidate concretization; $s \neq s_w \in [s_\mathcal{P}]$ is a witness of the next abstract transition $(s_\mathcal{P}, l, s'_\mathcal{P})$.

In the standard case (STD), the transition from $s_\mathcal{P}$ to $s'_\mathcal{P}$ is spurious because there is no corresponding concrete transition from the state s reached by a candidate concretization of the considered abstract path prefix. (More generally, this kind of spuriousness occurs if all states s reachable from ϕ_0, under all possible concretizations of the abstract path prefix, are not witnesses of $(s_\mathcal{P}, l, s'_\mathcal{P})$.)

In the new case (POL), in contrast, the required transition from s is there, s *is* a witness for $(s_\mathcal{P}, l, s'_\mathcal{P})$. Yet π does not select the required label, $\pi(s) \neq l$, although it does from a different state $s \neq s_w \in [s_\mathcal{P}]$. *The spuriousness here is exclusively due to policy decisions, i.e., to NN behavior.*

Ultimately, addressing this source of spuriousness requires the selection of new predicates characterizing π's decision boundary, i.e., where the NN does vs. does not select the label l within $s_\mathcal{P}$. Precise characterizations of this decision boundary remain a challenge for future work. For now, we approximate it with simple box constraint predicates that allow to distinguish the specific states s and s_w. Note that this approach is agnostic to the NN structure, which can be a key advantage if NN is large.

We remark that one can view $\pi(s) = l$ as a complex transition guard and apply standard CEGAR refinement techniques. This can be implemented by encoding the NN π into standard verification languages. Yet standard tools are not designed to handle such complex guards (one disjunction per neuron), and indeed our experiments with NUXMV indicate that CEGAR is usually not feasible on such encodings.

4.2 Refinement Methods

We introduce two different methods for refining $\mathcal{P}$ in the presence of policy-induced spuriousness as per Figure 2 (POL). Figure 3 illustrates these methods.

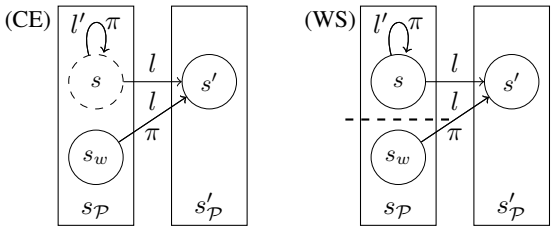

Figure 3: Illustration of concretization exclusion (CE) and witness splitting (WS). s and s_w as in Figure 2 (POL).

Our first method **concretization exclusion** (CE) focuses entirely on the state s in which π disagrees with the abstract transition $(s_{\mathcal{P}}, l, s'_{\mathcal{P}})$. We introduce predicates allowing to distinguish s from every other state in the state space. In other words, we approximate the complex transition guard $\pi(s) = l$ with $\neg s$. Specifically, we augment $\mathcal{P}$ with the box constraints $\{v \leq s(v) - 1, s(v) + 1 \leq v \mid v \in \mathcal{V}\}$.

Witness splitting (WS) is more targeted, by taking the witness s_w of $(s_{\mathcal{P}}, l, s'_{\mathcal{P}})$ into account. We introduce predicates allowing to distinguish s, not from every other state, but specifically from s_w. Intuitively, we approximate the decision boundary of π within $s_{\mathcal{P}}$ by the difference between s and s_w. We do so via box constraints of the form $v < s_w(v)$ where $s(v) < s_w(v)$, and $v > s_w(v)$ where $s(v) > s_w(v)$.

The box constraints used in both methods are motivated by their simplicity (facilitating effective handing).

5 CEGAR: Algorithm and Completeness

We now spell out our CEGAR algorithm for policy predicate abstraction. Specifically, we spell out a `refine` algorithm to be used in a loop around Vea22's method computing the fragment of $\Theta^{\pi}_{\mathcal{P}}$ reachable from ϕ_0 (more generally, around any method deciding reachability of ϕ_u from ϕ_0 in $\Theta^{\pi}_{\mathcal{P}}$) starting with an empty set of predicates $\mathcal{P} = \emptyset$.

If ϕ_u is not reachable, CEGAR stops with result "safe". Otherwise, `refine` is invoked, with an abstract unsafe path as input. If `refine` determines that path to be non-spurious, then CEGAR stops with result "unsafe". If the path is determined to be spurious, then `refine` adds new predicates to $\mathcal{P}$ and CEGAR iterates.

We next spell out the refinement algorithm, then prove that our policy predicate abstraction CEGAR is complete.

5.1 Refinement Algorithm

Consider Algorithm 1. Note first that the input is only the start state $s^0_{\mathcal{P}}$ of the unsafe abstract path, along with the underlying operator path $\langle o^0, \ldots, o^{n-1} \rangle$ – rather than the entire unsafe abstract path $\langle s^0_{\mathcal{P}}, \ldots, s^n_{\mathcal{P}} \rangle$ found by policy predicate abstraction. The latter would also be possible, as an alternative basis for the refinement step. Using only the operator path is a stronger form of refinement, as the same $\langle o^0, \ldots, o^{n-1} \rangle$ may underly mutiple unsafe abstract paths.

The algorithm starts by checking standard spuriousness (Figure 2 (STD)), as a cheap form of refinement covering transition behavior. We incrementally check whether there exists a start state from which a prefix of the operator path can be taken (line 2). Such path existence checks can be encoded as cheap SMT queries, similar to the abstract transition problem in standard predicate abstraction. If the check fails for some prefix, we refine with respect to the smallest spurious prefix $\langle o^0, \ldots, o^{i-1} \rangle$ (line 3), for the spuriousness causing constraint g^i (which we define to be ϕ_u for $i = n$).

We refine based on weakest preconditions as specified by procedure `WP` (line 17). Here, $wp_u(\phi)$ denotes the *syntactical weakest precondition* of ϕ applying update u, which is computed by substituting each $v \in dom(u)$ in ϕ by $u(v)$. In essence, `WP` computes the set $\{\phi^0_{wp}, \ldots, \phi^i_{wp}\}$ where ϕ^j_{wp} is the weakest precondition of ϕ taking $\langle o^j, \ldots, o^{i-1} \rangle$. This enables the refined abstraction to explicitly trace the truth value of ϕ along the entire path $\langle o^0, \ldots, o^{i-1} \rangle$.

Our implementation somewhat diverges from this procedure in that it splits formulas (Boolean combinations of constraints, as produced by the `WP` procedure) into their atoms. This is because the Vea22 implementation of policy predicate abstraction supports only atomic predicates (motivated by some of their algorithmic optimizations). Obviously, as

Algorithm 1: `refine` (Abstraction Refinement).

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 standard spuriousness.`

1 **for** $i \in \{0, \ldots, n\}$ **do**
2 **if** $\neg\exists \langle s^0, o^0, \ldots, o^{i-1}, s^i \rangle \in \Theta : s^0 \in [s^0_{\mathcal{P}}] \wedge s^0 \models \phi_0 \wedge s^i \models g^i$ **then**
3 $\mathcal{P} \leftarrow \mathcal{P} \cup$ `WP` $(g^i, \langle o^0, \ldots, o^{i-1} \rangle)$
4 **return** *SPURIOUS*

`// Check π-spuriousness.`

5 **let** $\langle s^0, o^0, \ldots, o^{n-1}, s^n \rangle \in \Theta : s^0 \in [s^0_{\mathcal{P}}] \wedge s^0 \models \phi_0 \wedge s^n \models \phi_u$ **in**
6 **for** $i \in \{0, \ldots, n-1\}$ **do**
7 **if** $\pi(s^i) \neq l^i$ **then**
8 **if** π-*Refinement* = "witness splitting" **then**
9 **let** $s^i_w \in [s^i_{\mathcal{P}}]$ **such that**
10 $s^i_w[\![o^i]\!] \in [s^{i+1}_{\mathcal{P}}] \wedge \pi(s^i_w) = l^i$ **in**
11 $\phi_{appr} \leftarrow$ `WitSplit` (s^i, s^i_w)
12 **else**
13 $\phi_{appr} \leftarrow \neg s^i$
14 $\mathcal{P} \leftarrow \mathcal{P} \cup$ `WP` $(\phi_{appr}, \langle o^0, \ldots, o^{i-1} \rangle)$
15 **return** π-*SPURIOUS* for s^0
16 **return** *NON-SPURIOUS*

17 **Procedure** `WP` $(\phi, \langle o^0, \ldots, o^{i-1} \rangle)$**:**
18 $\phi^i_{wp} \leftarrow \phi$
19 **for** $j \in \{i-1, \ldots, 0\}$ **do**
20 $\phi^j_{wp} \leftarrow wp_{u^j}(\phi^{j+1}_{wp})$
21 **return** $\{\phi^0_{wp}, \ldots, \phi^i_{wp}\}$

22 **Procedure** `WitSplit` (s, s_w)**:**
23 $\phi \leftarrow 1$
24 **for** $v \in \mathcal{V}$ *s.t.* $s(v) \neq s_w(v)$ **do**
25 **if** $s(v) < s_w(v)$ **then** $\phi \leftarrow \phi \wedge v < s_w(v)$
26 **if** $s(v) > s_w(v)$ **then** $\phi \leftarrow \phi \wedge v > s_w(v)$
27 **return** ϕ

new predicates, the collection of atoms allows to make all distinctions made by the overall formula.

If the checks for standard spuriousness result in a complete concrete path $\langle s^0, o^0, \ldots, o^{n-1}, s^n\rangle$, then we check that path for policy-induced spuriousness (Figure 2 (POL)), simply by evaluating the policy on each state and checking whether the desired label is selected. If yes (line 16), we have found a concrete unsafe path, thus proving unsafety. If no (line 7), the path is *π-spurious for s^0*. We then refine with respect to the shortest π-spurious prefix (line 14), using either witness splitting (line 11) or concretization exclusion (line 13). For the case of concretization exclusion, Algorithm 1 specifies a simple variant that suffices for completeness; our implementation uses $\{v \leq s(v) - 1, s(v) + 1 \leq v \mid v \in \mathcal{V}\}$ as specified in Section 4.2 because the Vea22 implementation does not support $=$ (respectively $\neq$) predicates.

Importantly, observe that, since we only consider a single concrete path $\langle s^0, o^0, \ldots, o^{n-1}, s^n\rangle$, the π-spuriousness check is incomplete: even if $\langle s^0, o^0, \ldots, o^{n-1}, s^n\rangle$ is π-spurious, there may exist a different non-π-spurious unsafe path using $\langle o^0, \ldots, o^{n-1}\rangle$ (starting from another state $s^0 \neq s \in [s^0_\mathcal{P}]$). For a complete test, we would need to encode the existence of such a path into a global SMT test, i.e., an SMT formula containing n copies of the neural network π. We do not pursue that option and instead settle for the cheap test in Algorithm 1. This source of incompleteness in our CEGAR refinement step is, however, counteracted by the iterative refinement of predicates, so that, as we show next, the overall algorithm is still complete.

5.2 Completeness

Theorem 1 (Completeness)**.** For any predicate set $\mathcal{P}$, CEGAR with Algorithm 1 will in finitely many iterations either prove unsafety or safety, i.e., either find a concrete unsafe path from ϕ_0 to ϕ_u, or terminate with a predicate set for which no abstract unsafe path exists.

To facilitate readability, we move the formal proof to the appendix. In the following, we give an overview of the *progress guarantees* provided by each of the refinement methods (i.e., standard, concretization exclusion and witness splitting), and we sketch the proof of Theorem 1. For all these methods, the weakest precondition computation $\texttt{WP}(\phi, \langle o^0, \ldots, o^{i-1}\rangle)$ constitutes the key element towards progress. It enables to explicitly trace the truth value of ϕ in the refined abstraction when taking $\langle o^0, \ldots, o^{i-1}\rangle$. Therefore, spurious abstract paths $\langle s^0_\mathcal{P}, o^0, \ldots, o^{i-1}, s^i_\mathcal{P}\rangle$ must be spurious for a reason different from $s^i_\mathcal{P} \models \phi$. For standard refinement, the progress guarantee is as follows:

Lemma 2 (Standard)**.** Let $\{\phi^0_{wp}, \ldots, \phi^i_{wp}\} \subseteq \mathcal{P}$ as computed by $\texttt{WP}(\phi, \langle o^0, \ldots, o^{i-1}\rangle)$ and let $\langle s^0_\mathcal{P}, o^0, \ldots, o^{i-1}, s^i_\mathcal{P}\rangle$ in $\Theta_\mathcal{P}$ with $s^i_\mathcal{P} \models \phi$. For any $\langle s^0, o^0, \ldots, o^{i-1}, s^i\rangle$ in Θ with $s^0 \in [s^0_\mathcal{P}]$ we have $s^i \models \phi$.

Lemma 2 guarantees that – after the refinement step – if there still exists a spurious abstract path $\sigma = \langle s^0_\mathcal{P}, o^0, \ldots, o^{i-1}, s^i_\mathcal{P}\rangle$ with $s^i_\mathcal{P} \models \phi$, then the spuriousness is due to a strict sub-prefix of σ. This spurious prefix can be detected and removed in subsequent CEGAR iterations. Since each iteration strictly decreases the size of the prefix, it is removed completely in finitely many steps.[2]

In what follows, we denote $\pi(s_\mathcal{P}) = \{\pi(s) \mid s \in [s_\mathcal{P}]\}$.

Lemma 3 (Concretization Exclusion)**.** Let $l \in \mathcal{L}$, $\langle s^0, o^0, \ldots, o^{i-1}, s^i\rangle$ in Θ with $\pi(s^i) \neq l$ and $\{\phi^0_{wp}, \ldots, \phi^i_{wp}\} \subseteq \mathcal{P}$ as computed by $\texttt{WP}(\neg s^i, \langle o^0, \ldots, o^{i-1}\rangle)$. For any $\langle s^0_\mathcal{P}, o^0, \ldots, o^{i-1}, s^i_\mathcal{P}\rangle$ in $\Theta_\mathcal{P}$ with $l \in \pi(s^i_\mathcal{P})$ we have $s^0 \notin [s^0_\mathcal{P}]$.

In words, s^i (respectively the path to s^i) is excluded from the set of concretizations for $\langle s^0_\mathcal{P}, o^0, \ldots, o^{i-1}, s^i_\mathcal{P}\rangle$ with $l \in \pi(s^i_\mathcal{P})$. Since in each iteration we remove at least one path from the set of possible concretizations, within finitely many iterations we either find a concretization that is valid under π, or remove the abstract (spurious) path completely.

Lemma 4 (Witness Splitting)**.** Let $\langle s^0, o^0, \ldots, o^{i-1}, s^i\rangle$ in Θ and $\{\phi^0_{wp}, \ldots, \phi^i_{wp}\} \subseteq \mathcal{P}$ as computed by $\texttt{WP}(\texttt{WitSplit}(s^i, s^i_w), \langle o^0, \ldots, o^{i-1}\rangle)$ for $s^i \neq s^i_w \in \mathcal{S}$. For any $\langle s^0_\mathcal{P}, o^0, \ldots, o^{i-1}, s^i_\mathcal{P}\rangle$ in $\Theta_\mathcal{P}$ with $s^i_w \in [s^i_\mathcal{P}]$ we have $s^0 \notin [s^0_\mathcal{P}]$.

In words, the path to s^i is excluded from the set of concretizations for $\langle s^0_\mathcal{P}, o^0, \ldots, o^{i-1}, s^i_\mathcal{P}\rangle$ with witness $s^i_w \in [s^i_\mathcal{P}]$. Again, in each iteration we split at least one state from the set of possible concretizations.

Proof sketch of Theorem 1. We show that Algorithm 1 strictly refines the abstraction, in the following sense: given the refined predicate set $\mathcal{P}' = \mathcal{P} \cup \{\phi^0_{wp}, \ldots, \phi^i_{wp}\}$, there exist concrete states $s, s' \in \mathcal{S}$ such that $s|_{\mathcal{P}'} \neq s'|_{\mathcal{P}'}$ while $s|_\mathcal{P} = s'|_\mathcal{P}$. Hence each refinement step distinguishes at least two new (non-empty) abstract states, $s|_{\mathcal{P}'}$ and $s'|_{\mathcal{P}'}$, and thus the number of iterations is finite.

Let $s^0_w, \ldots, s^i_w$ be the witness trace of abstract prefix $\langle s^0_\mathcal{P}, o^0, \ldots, o^{i-1}, s^i_\mathcal{P}\rangle$. Suppose $s^0_w, \ldots, s^i_w$ is no longer a valid witness trace under $\mathcal{P}'$. Then, for some j, $s^j_w[\![o^j]\!]|_{\mathcal{P}'} \neq s^{j+1}_w|_{\mathcal{P}'}$ while $s^j_w[\![o^j]\!]|_\mathcal{P} = s^{j+1}_w|_\mathcal{P}$. Hence $s = s^j_w[\![o^j]\!]$ and $s' = s^{j+1}_w$ satisfy the claim. Now suppose that $s^0_w, \ldots, s^i_w$ is still a valid witness trace. Then we can invoke Lemma 2, 3 or 4, according to which refinement step was used. This allows us to conclude that $s^0|_{\mathcal{P}'} \neq s^0_w|_{\mathcal{P}'}$ while $s^0|_\mathcal{P} = s^0_w|_\mathcal{P}$, where s^0 is the start state of the concretization (prefix) found by Algorithm 1 for $s^0_\mathcal{P}$ and $\langle o^0, \ldots, o^{i-1}\rangle$. Hence $s = s^0$ and $s' = s^0_w$ satisfy the claim in this case. □

6 Algorithmic Enhancements

The vanilla configuration of CEGAR– running a complete policy predicate abstraction from scratch in every iteration – is wasteful in at least two respects, that we address with algorithmic enhancements in our tool implementation.

[2] A more aggressive refinement approach would include all guards during the weakest precondition computation, removing the entire spurious path within a single iteration. Preliminary experiments indicate that this adds too many non-relevant predicates, hampering performance significantly.

Heuristic Search. Within CEGAR, policy predicate abstraction can stop as soon as the first unsafe abstract path is found. We hence employ heuristic search to find such paths quickly. This requires a heuristic function h mapping abstract states $s_\mathcal{P}$ to an estimate $h(s_\mathcal{P})$ of the distance to ϕ_u in $\Theta^\pi_\mathcal{P}$. We have so far instantiated this as follows:

Definition 5 (Hamming Distance)**.** Let $\mathcal{P}^0_u = \{p \in \mathcal{P} \mid \phi_u \vdash \neg p\}$ and $\mathcal{P}^1_u = \{p \in \mathcal{P} \mid \phi_u \vdash p\}$. The **hamming distance** of abstract state $s_\mathcal{P} \in \mathcal{S}_\mathcal{P}$ is $h_d(s_\mathcal{P}) = \big|\{p \in \mathcal{P}^0_u \mid s_\mathcal{P}(p) = 1\}\big| + \big|\{p \in \mathcal{P}^1_u \mid s_\mathcal{P}(p) = 0\}\big|$.

This heuristic function h_d counts the number of predicates for which $s_\mathcal{P}$ differs from the truth value entailed by the unsafety condition ϕ_u. We use h_d in a standard greedy best-first search (which expands search nodes by order of increasing heuristic value). As our experiments show, this simple method often improves performance drastically.

We remark that, although this idea is very natural, we are not aware of prior work using heuristic search in CEGAR for verification purposes.

Incremental Computation of $\Theta^\pi_\mathcal{P}$. An enhancement that has previously been explored in CEGAR is incremental computation (e.g., (Henzinger et al. 2002)). Rather than starting from scratch, we can reuse the transition information of already computed coarser abstractions:

Proposition 6. Given predicate sets $\mathcal{P} \subseteq \mathcal{P}'$ and abstract states $s_\mathcal{P} \subseteq s_{\mathcal{P}'}, s'_\mathcal{P} \subseteq s'_{\mathcal{P}'}$ then:

1. $(s_{\mathcal{P}'}, l, s'_{\mathcal{P}'}) \notin \mathcal{T}^\pi_{\mathcal{P}'}$, if $(s_\mathcal{P}, l, s'_\mathcal{P}) \notin \mathcal{T}^\pi_\mathcal{P}$.
2. $(s_{\mathcal{P}'}, l, s'_{\mathcal{P}'}) \in \mathcal{T}^\pi_{\mathcal{P}'}$, if $(s_\mathcal{P}, l, s'_\mathcal{P}) \in \mathcal{T}^\pi_\mathcal{P}$ with witness $(s_w, l, s') \in \mathcal{T}^\pi$ such that $s_w \in [s_{\mathcal{P}'}]$ and $s' \in [s'_{\mathcal{P}'}]$.

Due to (1.), we can skip checks for abstract transitions not possible in coarser abstractions. Due to (2.), we can reuse the transition witnesses computed in coarser abstractions.

7 Experiments

We implemented our CEGAR approach on top of Vea22's C++ code base. We fix Vea22's best-performing configuration of policy predicate abstraction. For the standard spuriousness check (Algorithm 1, line 2) we query $Z3$ (de Moura and Bjørner 2008). [3] 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. We next describe the experiments setup, then summarize our results.

7.1 Experiments Setup

CEGAR Configurations. Witness splitting is denoted **WS**, concretization exclusion **CE**. To provide an ablation study, we also run **WS** (which tends to perform best) without heuristic search in ~~h_d~~**-WS**, and without incremental computation in ~~**inc**~~**-WS**. All configurations start with $\mathcal{P} = \emptyset$.

Competitors in NUXMV. We compare against a broad range of state-of-the-art verification methods implemented in **NUXMV** (Cavada et al. 2014), by encoding the neural policy π into constraints in NUXMV's input language. A description of the encoding is available in the appendix. We experiment with: bounded model checking (**BMC**) (Biere et al. 1999) and simple bounded model checking (**SBMC**) (Biere et al. 2006), both using SMT instead of SAT; implicit predicate abstraction (**IPA**) (Tonetta 2009), running BMC with k-induction (Sheeran, Singh, and Stålmarck 2000) within a CEGAR loop; explicit predicate abstraction within a CEGAR loop (**EPA**); an SMT-based cone of influence (**COI**) algorithm, a form of CEGAR over projections onto iteratively larger variable subsets (see, e.g., (Clarke, Grumberg, and Peled 2001)); as well as NUXMV's **IC3** (Cimatti and Griggio 2012). Except for BMC, all these algorithms are complete, i.e., can prove safety.

Benchmarks. We use Vea22's benchmarks, except for Racetrack which has a polynomial-size state space and is easily tackled by naïve explicit enumeration. To give a brief overview, the benchmarks are variants of the planning domains Blocksworld, SlidingTiles and Transport. In the former two domains, actions moving a block/tile x may probabilistically fail, in which case the cost of moving x (represented by a state variable) is incremented. These probabilistic transitions (over which the policy is learned) are abstracted into non-deterministic ones for the purpose of verification, amounting to a worst-case analysis. In all domains, the start conditions impose a partial order on the block/tile positions. In Blocksworld, a state is unsafe if the number of blocks on the table exceeds a fixed limit. In SlidingTiles, unsafe states are specified in terms of a set of unsafe tile positions. In Transport, a truck must deliver packages while safely crossing a bridge with limited capacity.

For each domain instance, there are three NN policies trained by Vea22 using Q-learning (Mnih et al. 2015), each with 2 hidden layers of size 16, 32, respectively 64. On Blocksworld and SlidingTiles, there are policies that do, and ones that do not, take move costs into account.

7.2 Experiments Results

Table 1 shows our results. The conclusions are:

CE vs. WS. For policy-induced spuriousness, WS is generally the better refinement method than CE. While CE outperforms WS on 11 problem instances, this pertains to easier instances and WS scales up further (solving 3 more instances). This makes sense as the rather aggressive refinement strategy pursued by CE is feasible only on smaller instances, where it terminates more quickly than the more cautious strategy taken by WS.

Ablation Study. Heuristic search improves performance drastically, often by orders of magitude. WS succeeds in 7 more instances than ~~h_d~~-WS, outperforming it on all benchmarks except Transport. The impact of incremental computation is less drastic, but it nevertheless helps, with WS having better runtime than ~~inc~~-WS in 17 cases, albeit often to small extents. We remark that the impact is much higher when not using heuristic search, i.e., that enhancement somewhat overshadows incremental computation.

[3] Our tool (and all experiments) are publicly available (https://fai.cs.uni-saarland.de/vinzent/downloads/aaai23.zip) .

Benchmark	NN	Safe	CEGAR Configurations				NUXMV Configurations				WS			
			WS	~~h_d~~-WS	~~inc~~-WS	CE	BMC	SBMC	IPA	COI	$\|\sigma\|$	$\|P\|$	# It.	π-It.
	16	✓	8.6	12.6	10.9	**4.1**	-	-	-	-		19	10	3
4 Blocks	32	✓	13.4	22.3	14.2	**6.9**	-	-	-	-		20	9	3
(cost-ignoring)	64	✓	81.9	310.4	112.5	**53.5**	-	-	-	-		20	10	2
	16	✓	**255.3**	959.5	302.7	360.0	-	-	-	-		40	23	4
6 Blocks	32	✓	280.7	3659.5	337.2	**208.6**	-	-	-	-		38	18	5
(cost-ignoring)	64	✓	5934.2	-	8929.9	**1147.8**	-	-	-	-		34	19	7
	16	✓	29155.7	-	**20036.0**	31234.4	-	-	-	-		64	32	7
8 Blocks	32	?	-	-	-	-	-	-	-	-		63	36	4
(cost-ignoring)	64	?	-	-	-	-	-	-	-	-		14	5	1
	16	×	280.5	3936.3	300.9	221.5	**16.5**	38.0	-	26.6	2	50	26	18
8 Puzzle	32	✓	39868.1	-	42098.4	**31168.2**	-	-	-	-		123	72	14
(cost-ignoring)	64	✓	**42285.7**	-	42813.3	-	-	-	-	-		128	66	12
	16	✓	370.4	442.6	378.8	**88.5**	-	-	-	-		26	11	4
4 Blocks	32	✓	604.2	827.5	5041.1	**599.1**	-	-	-	-		26	11	5
(cost-aware)	64	✓	**25081.5**	-	25893.1	-	-	-	-	-		29	10	4
	16	✓	**2413.2**	-	2524.1	27314.0	-	-	-	-		42	21	6
6 Blocks	32	✓	**1323.7**	8650.9	1344.6	-	-	-	-	-		42	17	5
(cost-aware)	64	?	-	-	-	-	-	-	-	-		26	6	2
	16	×	169.2	28756.8	207.5	10.0	11.1	**7.8**	1052.5	25.1	2	63	13	8
8 Blocks	32	?	-	-	-	-	-	-	-	-		63	20	8
(cost-aware)	64	?	-	-	-	-	-	-	-	-		21	4	1
	16	×	8039.2	-	9342.9	7383.4	4609.7	**811.4**	-	2740.9	4	87	30	15
8 Puzzle	32	?	-	-	-	-	-	-	-	-		86	34	13
(cost-aware)	64	?	-	-	-	-	-	-	-	-		79	25	11
	16	×	1.3	**1.2**	**1.2**	335.4	245.7	2283.0	-	2194.4	1	16	4	3
Transport	32	×	**63.7**	68.0	67.4	360.7	1500.1	38549.0	-	43190.1	1	39	22	20
	64	×	**1.4**	**1.4**	**1.4**	608.6	-	15.7	-	112.0	1	12	3	2

Table 1: Runtime results in seconds for the evaluated CEGAR configurations (WS, ~~h_d~~-WS, ~~inc~~-WS, CE) and NUXMV competitors (BMC, SBMC, IPA, COI) over different benchmarks and NN policies (distinguishing cost-aware policies and cost-ignoring policies where applicable). The NUXMV competitors IC3 and EPA did not terminate for any instance and are therefore omitted. - indicates timeouts (exceeding the 12h time limit). For WS we additionally provide the length of the unsafe path $|\sigma|$ (if found), the size of the final predicate set ($|\mathcal{P}|$), the number of CEGAR iterations (# It.) and the number of iterations with π-spuriousness refinement (π-It.). $|\sigma|$ agrees with the unsafe path lengths of all NUXMV competitors.

Comparison against the state of the art in NUXMV. Our key result is that *our approach tends to outperform the (non-NN-tailored) state of the art in* NUXMV. *In particular, ours is the only approach in our experiments that succeeds in proving policies safe.* Although all NUXMV algorithms except BMC can prove safety in principle, they never succeed in doing so here. On the 6 problem instances shown to be unsafe, the picture is more mixed. On 3 of these, at least one of the NUXMV competitors is faster than our approach, by 1-2 orders of magnitude. In Transport though, our methods are 2 orders of magnitude faster than the best NUXMV competitor BMC (we remark that the length of the unsafe paths here is small – all tools find paths of the same length as specified for WS in column $|\sigma|$). NUXMV IC3 and CEGAR do not terminate for any instance at all. The bad performance of EPA and IPA indicates strongly that non-NN-tailored CEGAR techniques are not feasible for neural policy verification.

CEGAR statistics for WS. On the right-hand side of Table 1, we shed some light on the scale of predicate sets and CEGAR processes (data shown for non-terminated runs is the largest number reached). As one would expect, the final number of predicates is typically correlated with instance scale (low numbers for non-terminated runs are due to slow progress). The data also indicates a current scalability limit around 80 predicates, though there are exceptions. The number of CEGAR iterations behaves similarly. Standard transition-spuriousness refinement dominates, which makes sense when starting from $\mathcal{P} = \emptyset$ where a sensible abstraction must be built up in the first place. In Transport though, almost all refinement steps pertain to π-spuriousness. Presumably this is due to repeated iterations to find the correct start state of the final non-π-spurious unsafe path.

8 Conclusion

The verification of neural network behavior is important. Here we contribute a CEGAR method dealing with the new source of spuriousness induced by policy decisions, thus fully automating verification via policy predicate abstraction. Our experimental results are highly encouraging, vastly outperforming non-NN-tailored state-of-the-art algorithms in the ability to prove safety on a collection of benchmarks.

The sky is the limit for further extensions of this approach. One should be able to obtain much better heuristic functions, and partial safety verification is possible by continuing CEGAR on instances already proved to be unsafe. Probabilistic transition systems, infinite transition systems, continuous-state transition systems can all in principle be handled. A major step will be to support more advanced NN structures such as graph neural networks (Toyer et al. 2020; Stahlberg, Bonet, and Geffner 2022).

Acknowledgments

This work was funded by DFG Grant 389792660 as part of TRR 248 – CPEC (https://perspicuous-computing.science). This work has received funding from the European Union's Horizon Europe Research and Innovation program under the grant agreement TUPLES No 101070149.

References

Akintunde, M.; Lomuscio, A.; Maganti, L.; and Pirovano, E. 2018. Reachability Analysis for Neural Agent-Environment Systems. In Thielscher, M.; Toni, F.; and Wolter, F., eds., *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 *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*, 193–203. IEEE.

Ball, T.; Majumdar, R.; Millstein, T. D.; and Rajamani, S. K. 2001. Automatic Predicate Abstraction of C Programs. In Burke, M.; and Soffa, M. L., eds., *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, 203–213. ACM.

Barrett, C. W.; and Tinelli, C. 2018. Satisfiability Modulo Theories. In Clarke, E. M.; Henzinger, T. A.; Veith, H.; and Bloem, R., eds., *Handbook of Model Checking*, 305–343. Springer.

Biere, A.; Cimatti, A.; Clarke, E. M.; and Zhu, Y. 1999. Symbolic Model Checking without BDDs. In Cleaveland, R., ed., *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *LNCS*, 193–207. Springer.

Biere, A.; Heljanko, K.; Junttila, T. A.; Latvala, T.; and Schuppan, V. 2006. Linear Encodings of Bounded LTL Model Checking. *Log. Methods Comput. Sci.*, 2(5).

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 Biere, A.; and Bloem, R., eds., *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.

Cimatti, A.; and Griggio, A. 2012. Software Model Checking via IC3. In Madhusudan, P.; and Seshia, S. A., eds., *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *LNCS*, 277–293. 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.

Clarke, E. M.; Grumberg, O.; and Peled, D. A. 2001. *Model checking, 1st Edition*. MIT Press.

de Moura, L.; and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In Ramakrishnan, C. R.; and Rehof, J., eds., *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008 Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *LNCS*, 337–340. Springer.

Dutta, S.; Chen, X.; and Sankaranarayanan, S. 2019. Reachability analysis for neural feedback systems using regressive polynomial rule inference. In Ozay, N.; and Prabhakar, P., eds., *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 Benton, J.; Lipovetzky, N.; Onaindia, E.; Smith, D. E.; and Srivastava, S., eds., *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 Grumberg, O., ed., *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 de Weerdt, M.; Koenig, S.; Röger, G.; and Spaan, M. T. J., eds., *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.

Gupta, A.; and Strichman, O. 2005. Abstraction Refinement for Bounded Model Checking. In Etessami, K.; and Rajamani, S. K., eds., *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *LNCS*, 112–124. Springer.

Henzinger, T. A.; Jhala, R.; Majumdar, R.; and McMillan, K. L. 2004. Abstractions from proofs. In Jones, N. D.; and Leroy, X., eds., *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, 232–244. ACM.

Henzinger, T. A.; Jhala, R.; Majumdar, R.; and Sutre, G. 2002. Lazy abstraction. In Launchbury, J.; and Mitchell,

J. C., eds., *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, 58–70. ACM.

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 de Weerdt, M.; Koenig, S.; Röger, G.; and Spaan, M. T. J., eds., *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.; 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 Dillig, I.; and Tasiran, S., eds., *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.

Podelski, A.; and Rybalchenko, A. 2007. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In Hanus, M., ed., *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007*, volume 4354 of *LNCS*, 245–259. Springer.

Sheeran, M.; Singh, S.; and Stålmarck, G. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In Jr., W. A. H.; and Johnson, S. D., eds., *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *LNCS*, 108–125. Springer.

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

Sun, X.; Khedr, H.; and Shoukry, Y. 2019. Formal Verification of Neural Network Controlled Autonomous Systems. In Ozay, N.; and Prabhakar, P., eds., *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*, 147–156. ACM.

Tonetta, S. 2009. Abstract Model Checking without Computing the Abstraction. In Cavalcanti, A.; and Dams, D., eds., *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *LNCS*, 89–105. Springer.

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.; Steinmetz, M.; and Hoffmann, J. 2022. Neural Network Action Policy Verification via Predicate Abstraction. In Kumar, A.; Thiébaux, S.; Varakantham, P.; and Yeoh, W., eds., *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling, ICAPS 2022, Singapore (virtual), June 13-24 2022*. AAAI Press.

A Proofs

In this section, we attach the proof of the completeness result stated in the main text. In a first step we formally show that the weakest precondition set $\{\phi^0_{wp}, \ldots, \phi^i_{wp}\}$ as computed by $\texttt{WP}(\phi, \langle o^0, \ldots, o^{i-1}\rangle)$ enables to trace the truth value of ϕ taking $\langle o^0, \ldots, o^{i-1}\rangle$. Subsequently, we use this to prove the lemmas in the main text.

Lemma A (WP)**.** Let $\{\phi^0_{wp}, \ldots, \phi^i_{wp}\} \subseteq \mathcal{P}$ as computed by $\texttt{WP}(\phi, \langle o^0, \ldots, o^{i-1}\rangle)$ (i.e., $\phi^i_{wp} = \phi$).

(a) For any $\langle s^0, o^0, \ldots, o^{i-1}, s^i\rangle$ in Θ it holds $s^0(\phi^0_{wp}) = s^i(\phi^i_{wp})$.

(b) For any $\langle s^0_{\mathcal{P}}, o^0, \ldots, o^{i-1}, s^i_{\mathcal{P}}\rangle$ in $\Theta_{\mathcal{P}}$ it holds $s^0_{\mathcal{P}}(\phi^0_{wp}) = s^i_{\mathcal{P}}(\phi^i_{wp})$.

Proof by induction on i. We use that for the syntactical weakest precondition $wp_u(\phi)$ with $o = (g, l, u)$, it holds for any state $s \in \mathcal{S}$:

$$s(wp_u(\phi)) = s[u(s)](\phi) = s[\![o]\!](\phi) \qquad (*)$$

Induction base ($i = 0$): Trivial.
Induction step ($i > 0$):

(a) By induction hypothesis $s^0(\phi^0_{wp}) = s^{i-1}(\phi^{i-1}_{wp})$. By (*) $s^{i-1}(\phi^{i-1}_{wp}) = s^i(\phi^i_{wp})$. Thus $s^0(\phi^0_{wp}) = s^i(\phi^i_{wp})$.

(b) By induction hypothesis $s^0_{\mathcal{P}}(\phi^0_{wp}) = s^{i-1}_{\mathcal{P}}(\phi^{i-1}_{wp})$. Moreover, let $s^{i-1} \in [s^{i-1}_{\mathcal{P}}]$ be a transition witness (i.e., $s^{i-1}[\![o^{i-1}]\!] \in [s^i_{\mathcal{P}}]$). It follows $s^{i-1}_{\mathcal{P}}(\phi^{i-1}_{wp}) = s^{i-1}(\phi^{i-1}_{wp}) \overset{(*)}{=} s^{i-1}[\![o^{i-1}]\!](\phi^i_{wp}) = s^i_{\mathcal{P}}(\phi^i_{wp})$. Thus $s^0_{\mathcal{P}}(\phi^0_{wp}) = s^i_{\mathcal{P}}(\phi^i_{wp})$.

□

We now use Lemma A to proof the lemmas from the main text.

Proof of Lemma 2. Since $s^i_{\mathcal{P}} \models \phi$, we have $s^i_{\mathcal{P}}(\phi^i_{wp}) = 1$. Hence by Lemma A (b) also $s^0_{\mathcal{P}}(\phi^0_{wp}) = 1$. Thus, since $s^0 \in [s^0_{\mathcal{P}}]$, $s^0(\phi^0_{wp}) = 1$ and therefore, by Lemma A (a), $s^i(\phi^i_{wp}) = 1$ respectively $s^i \models \phi$. □

Proof of Lemma 3. Since $l \in \pi(s^i_{\mathcal{P}})$ while $\pi(s^i) \neq l$ and $s^i|_{\mathcal{P}} = \{s^i\}$, we have $s^i \notin [s^i_{\mathcal{P}}]$, and thus $s_{\mathcal{P}}(\phi^i_{wp}) = 1$ (where $\phi^i_{wp} = \neg s^i$). Hence by Lemma A (b) also $s^0_{\mathcal{P}}(\phi^0_{wp}) = 1$. Moreover, since $s^i(\phi^i_{wp}) = 0$ and therefore, by Lemma A (a), $s^0(\phi^0_{wp}) = 0$, it follows $s^0 \notin [s^0_{\mathcal{P}}]$. □

Proof of Lemma 4. Since $s^i_w \in [s^i_{\mathcal{P}}]$, we have $s^i_{\mathcal{P}}(\phi^i_{wp}) = 1$ (where $\phi^i_{wp} = \texttt{WitSplit}(s^i, s^i_w)$). Hence by Lemma A (b) also $s^0_{\mathcal{P}}(\phi^0_{wp}) = 1$. Moreover, since $s^i(\phi^i_{wp}) = 0$ by Lemma A (a) it follows $s^0(\phi^0_{wp}) = 0$, and thus $s^0 \notin [s^0_{\mathcal{P}}]$. □

Finally, we proof Theorem 1

Proof of Theorem 1. Let $\mathcal{P}'$ be the predicate set obtained by refinement of $\mathcal{P}$ in an arbitrary iteration, i.e., $\mathcal{P}' = \mathcal{P} \cup \{\phi^0_{wp}, \ldots, \phi^i_{wp}\}$ computed as per WP according to the applied refinement case in Algorithm 1 (line 3 or 14). We show that there exist concrete states $s, s' \in \mathcal{S}$ such that $s|_{\mathcal{P}'} \neq s'|_{\mathcal{P}'}$ while $s|_{\mathcal{P}} = s'|_{\mathcal{P}}$. Then, with each refinement step distinguishing at least two new (non-empty) abstract states – namely $s|_{\mathcal{P}'}$ and $s'|_{\mathcal{P}'}$ – the number of iterations is upper bounded by $|\mathcal{S}|$.

In each refinement step, we consider a path prefix $\langle s^0_{\mathcal{P}}, o^0, \ldots, o^{i-1}, s^i_{\mathcal{P}}\rangle$ with witness trace $s^0_w, \ldots, s^i_w$ (i.e., $s^j_w \in [s^j_{\mathcal{P}}]$, $s^j_w[\![o^j]\!] \in [s^{j+1}_{\mathcal{P}}]$ and $\pi(s^j_w) = l^j$). Moreover, let $\langle s^0, o^0, \ldots, o^{i-1}, s^i\rangle$ be the concretization for $s^0_{\mathcal{P}}$ and $\langle o^0, \ldots, o^{i-1}\rangle$ as found by Algorithm 1 (line 2 at step $i-1$ respectively line 5).[4] Case distinction:

- Suppose $s^0_w, \ldots, s^i_w$ is not a valid witness trace under the refined predicate set $\mathcal{P}'$. That is, there exists $j \in \{0, \ldots, i-1\}$ such that $s^j_w[\![o^j]\!]|_{\mathcal{P}'} \neq s^{j+1}_w|_{\mathcal{P}'}$. Then, since on the contrary $s^j_w[\![o^j]\!]|_{\mathcal{P}} = s^{j+1}_w|_{\mathcal{P}}$, the statement holds.
- Suppose $s^0_w, \ldots, s^i_w$ is still a valid witness trace under $\mathcal{P}'$, specifically for $\langle s^0_{\mathcal{P}'}, o^0, \ldots, o^{i-1}, s^i_{\mathcal{P}'}\rangle$. Case distinction over the refinement methods:
  - (Standard) Case distinction:
    - ($s^0 \in [s^0_{\mathcal{P}'}]$) By contradiction. Since $s^i_w \models \phi^i_{wp}$, we also have $s^i_{\mathcal{P}'} \models \phi^i_{wp}$. Hence, by Lemma 2, for the path $\langle s^0, o^0, \ldots, o^{i-1}, s^i\rangle$ we have $s^i \models \phi^i_{wp}$. But then, since $s^0 \in [s^0_{\mathcal{P}'}] \subseteq [s^0_{\mathcal{P}}]$, this contradicts the positive spuriousness check in Algorithm 1 (line 2, at step i).
    - ($s^0 \notin [s^0_{\mathcal{P}'}]$) Then $s^0_w|_{\mathcal{P}'} \neq s^0|_{\mathcal{P}'}$ while $s^0_w|_{\mathcal{P}} = s^0|_{\mathcal{P}}$, and thus the statement holds.
  - (Concretization Exclusion) Since $\pi(s^i) \neq \pi(s^i_w) \in \pi(s^i_{\mathcal{P}'})$, by Lemma 3 we have $s^0 \notin [s^0_{\mathcal{P}'}]$ and thereby $s^0|_{\mathcal{P}'} \neq s^0_w|_{\mathcal{P}'}$ while $s^0|_{\mathcal{P}} = s^0_w|_{\mathcal{P}}$ and thus the statement holds.
  - (Witness Splitting) By Lemma 4 it holds $s^0 \notin [s^0_{\mathcal{P}'}]$ and thus $s^0|_{\mathcal{P}'} \neq s^0_w|_{\mathcal{P}'}$ while $s^0|_{\mathcal{P}} = s^0_w|_{\mathcal{P}}$ and thus the statement holds.

□

[4] Note, for $i = 0$ there still exists $s^0 \in [s^0_{\mathcal{P}}]$ with $s^0 \models \phi_0$.

B NUXMV

Using the NUXMV toolkit (Cavada et al. 2014), we can encode our model along with the neural action policy with various constraint and assignment statements. It is worth noting that NUXMV does not offer any optimisations built especially for neural networks with ReLU activation functions as in our proposed approach.

The various components of the state space and action policy are encoded in NUXMV as follows:

- The state variables $\mathcal{V}$ are directly encoded as bounded integer variables (respectively where possible as Booleans) in NUXMV.
- The start condition ϕ_0, the unsafety condition ϕ_u and operator guards g are encoded as constraints on the state variables (using the NUXMV initial condition command INIT for ϕ_0)
- Each operator update u is encoded as a conjunction of update equations, i.e., $next(v) = u(v)$ if $v \in dom(u)$ and $next(v) = v$ if $v \notin dom(u)$ for each $v \in \mathcal{V}$.
- The neural network's architecture is encoded by introducing an auxiliary variable for each neuron in the network. Let $v_{i,j}$ be the variable for neuron j in layer i. Let $w^i_{j,k}$ denote the weight connecting neuron j in layer i with neuron k in $i-1$, let b^i_j denote the bias of j in i, and let d_i denote the size of layer i. For each hidden layer neuron we add a constraint

  $$v_{i,j} = \max(w^i_{j,1} \cdot v_{i-1,1} + \cdots + w^i_{j,d_{i-1}} \cdot v_{i-1,d_{i-1}} + b^i_j, 0)$$

  with $\max(x, 0)$ encoding the ReLU activation. For each output layer neuron we add a constraint

  $$v_{i,j} = w^i_{j,1} \cdot v_{i-1,1} + \cdots + w^i_{j,d_{i-1}} \cdot v_{i-1,d_{i-1}} + b^i_j.$$

- The action label selection (argmax over the NN output) is encoded using greater-than constraints.
- Finally the transition relation (using the nuXmv TRANS command) is encoded as a case distinction over the label selection; with each case being a disjunction of the respectively labeled operators; and each operator being the conjunction of its guard and update.