Neural Policy Verification via Predicate Abstraction: CEGAR

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. Recent work has shown how to extend predicate abstraction to prove safety of such π, within an abstract state space over-approximating all possible behaviors. Specifically, this work employs *predicate abstraction*, where abstract states are characterized by the truth values of a set $\mathcal{P}$ of constraints (the *predicates*) over the state variables. Empirical results show that this approach is promising, outperforming explicit enumeration and bounded-length verification on a set of benchmarks with large non-deterministic state spaces. However, the sets $\mathcal{P}$ underlying these results were supplied manually. Here we show how to automate this step. We extend the well-known *counter-example guided abstraction refinement (CEGAR)* paradigm to the new setting. This involves dealing with a new source of spuriousness in counter-examples (abstract unsafe paths), pertaining not to transition behavior but to the decisions of the action policy π. We introduce two methods tackling this issue, along with a number of algorithmic optimizations. Our results show that automatic safety verification of NN action policies is feasible with this approach, thanks in particular to our optimizations which often yield dramatic performance benefits.

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 line of research explores the use of bounded model checking and k-induction for neural controller verification (Akintunde et al. 2018, 2019; Amir, Schapira, and Katz 2021).

Here we follow up on recent work by Vinzent et al. (2022) (henceforth: **Vea22**) verifying safety of NN action policies π taking discrete action choices in sequential decision making, specifically non-deterministic state spaces over bounded-integer state variables, via **predicate abstraction (PA)** (Graf and Saïdi 1997; Ball et al. 2001; Henzinger et al. 2004).

This approach builds an 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 $x \leq y$). Abstract states are characterized by truth value assignments to $\mathcal{P}$. Like in other abstraction methods known in planning (e.g. (Edelkamp 2001; Helmert et al. 2014; Seipp and Helmert 2018)), transitions are over-approximated to preserve all possible behaviors. However, the method 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 build the fragment of the **policy predicate abstraction (PPA)** $\Theta^\pi_\mathcal{P}$ – the predicate abstraction of Θ^π – reachable from ϕ_0, and check whether ϕ_u is reached. If this is not the case, then π is safe.

Vea22 run experiments on a collection of benchmarks with large non-deterministic state spaces, adapted from SlidingTiles, Blocksworld, and Transportation style problems, with NN policies trained by deep Q-learning (Mnih et al. 2015). Their empirical results show that PPA is promising, outperforming two baselines, namely explicit enumeration (enumerating the reachable fragment of Θ^π) and bounded-length verification (compiling length-L safety into an SMT formula (de Moura and Bjørner 2008)). However, the sets $\mathcal{P}$ underlying these results were supplied manually. Vea22 examined performance as a function of $|\mathcal{P}|$, with predicate sets scaled according to simple manually designed schemes. The positive results pertain to particular points in these scaling schemes: particular instances of $\mathcal{P}$ that work well, which are often located at sweetspots in the middle of the scaling scheme. Hence it remained unclear 1. how to find suitable $\mathcal{P}$ automatically, and 2. whether doing so incurs an infeasible overhead from computing abstract state spaces outside the sweetspot.

Here we show how to address 1. in a way answering 2. in the negative. We do so by extending the well-known *counter-example guided abstraction refinement (CEGAR)* (Clarke et al. 2003) paradigm to the new PPA setting. This involves dealing with a new source of spuriousness in counter-examples, i.e., abstract unsafe paths whose concretization fails. Namely, say that the concretization of the abstract unsafe path (processing that path step-by-step to find a concrete unsafe path) has so far lead to the concrete state s_c in the abstract state $s_\mathcal{P}$, and say that the next transition on the abstract unsafe path is $(s_\mathcal{P}, l, s'_\mathcal{P})$. Then, in standard predicate abstraction, the concretization of this transition fails iff there is no concrete transition (s_c, l, s') s.t. $s' \in s'_\mathcal{P}$. In PPA however, it can happen that such a transition does exist, yet the policy does not choose the right action, $\pi(s_c) \neq l$. In this case, *the source of spuriousness is the policy behavior (and not the transition behavior) in s_c.*

We introduce two methods tackling this issue: as a straightforward naïve solution, we introduce new predicates allowing to distinguish s_c from all other states; as a more targeted solution, we consider a state $s_w \in s_\mathcal{P}$ with transition (s_w, l, s') and $s' \in s'_\mathcal{P}$ as a witness for the abstract transition $(s_\mathcal{P}, l, s'_\mathcal{P})$, and introduce new predicates allowing to distinguish between s_c and s_w.

We also design a number of algorithmic optimizations to avoid computational overhead in CEGAR. In particular, we use heuristic search to find abstract counter-examples quickly, and thus avoid to build large parts of the abstract state space in all but the last iteration of CEGAR.

Our empirical results show that automatic safety verification of NN action policies is feasible with this approach, thanks in particular to our optimizations which often yield dramatic performance benefits.

Preliminaries

State Space Representation 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 is a non-empty bounded integer interval. We denote by Exp the set of **linear integer expressions** over $\mathcal{V}$, i.e., expressions 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 $e_1, e_2 \in Exp$, and all Boolean combinations thereof. 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** $u \colon \mathcal{V} \to Exp$.

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 $e \in Exp$ 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 $s' = s[u(s)]$, i.e., the guard is satisfied in the source state s, and the successor state s' results from applying the update to s. Here, $u(s)$ denotes the partial variable assignment induced by u evaluated over s, i.e., $u(s) = \{v \mapsto u(v)(s) \mid v \in dom(u)\}$.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).

NN Action Policies. An **action policy** π is a function $\mathcal{S} \to \mathcal{L}$. The **policy-restricted state space** Θ^π is the subgraph $\langle \mathcal{S}, \mathcal{L}, \mathcal{T}^\pi \rangle$ of Θ with $\mathcal{T}^\pi = \{(s, l, s') \in \mathcal{T} \mid \pi(s) = l\}$.

We consider action policies represented by **neural networks** (NN), specifically fully connected feed-forward NN. These consist of an input layer with an input for each state variable; arbitrarily many hidden layers; and an output layer with an output for each action. The policy π is obtained by applying argmax to the output layer. While the approach is, in principle, agnostic to the activation functions used, Vea22 leverage optimizations specific to (piecewise-linear) **rectified linear units (ReLU)**, $ReLU(x) = \max(x, 0)$.

Safety. 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. That is, Θ is **unsafe** with respect to ρ iff there exists a path $\langle s^0, o^0, s^1, o^1, \ldots, s^n, o^n, s^u \rangle$ in Θ such that $s^0 \models \phi_0$ and $s^u \models \phi_u$.[1] Otherwise Θ is **safe**.

The notion above transfers to the policy-restricted case in a straight-forward manner. That is, a policy π is safe with respect to ρ, iff Θ^π is safe with respect to ρ. Otherwise π is unsafe. That said, for the remainder of this chapter, we revisit predicate abstraction concepts for the verification of standard (non-policy-restricted) safety.

Predicate Abstraction. Predicate abstraction (Graf and Saïdi 1997) is a well-established abstraction technique from formal methods. 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 $s_\mathcal{P}$. Accordingly, we say that $s_\mathcal{P}$ satisfies a constraint $\phi \in C$, written $s_\mathcal{P} \models \phi$, iff there exists $s \in [s_\mathcal{P}]$ such that $s \models \phi$. The abstract state space now is defined in a transition-preserving manner:

Definition 1 (Predicate Abstraction)**.** The **predicate abstraction** of Θ over $\mathcal{P}$ is 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 predicates states over $\mathcal{P}$, and $\mathcal{T}_\mathcal{P} = \{(s|_\mathcal{P}, l, s'|_\mathcal{P}) \mid (s, l, s') \in \mathcal{T}\}$.

The computation of $\Theta_\mathcal{P}$ necessitates to solve 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 transition problems are routinely

[1]Throughout the paper we consider operator-specific paths as this is the path granularity used by our abstraction refinement.

addressed as satisfiability modulo theories (SMT) (Barrett et al. 1994) formulas, which can be answered using existing solvers (e.g. $Z3$ (de Moura and Bjørner 2008)).

Safety Verification. 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 $\langle s^0_{\mathcal{P}}, o^0, \ldots, o^n, s^u_{\mathcal{P}} \rangle$ in $\Theta_{\mathcal{P}}$ such that $s^0_{\mathcal{P}} \models \phi_0$, $s^u_{\mathcal{P}} \models \phi_u$. Otherwise $\Theta_{\mathcal{P}}$ is **safe**. Due to the over-approximating nature of $\Theta_{\mathcal{P}}$, safety in Θ can be proven via safety in $\Theta_{\mathcal{P}}$:

Proposition 2 (Safety in $\Theta_{\mathcal{P}}$)**.** Let ρ be a safety property. If $\Theta_{\mathcal{P}}$ is safe with respect to ρ, then so is Θ.

The opposite is not true, i.e., unsafety of $\Theta_{\mathcal{P}}$ does not imply unsafety of Θ. That is, an abstract (unsafe) path in $\Theta_{\mathcal{P}}$ may be spurious, i.e., without concretization in Θ. Here, counter example guided abstraction refinement (**CEGAR**) (e.g. (Clarke et al. 2000)) is a popular approach to refine $\mathcal{P}$ to iteratively remove such spurious abstract paths, until either the abstraction is proven safe, or a non-spurious abstract path is found, hence certifying unsafety of Θ. CEGAR is applicable, in principle, as long as a technique for abstraction refinement exists (e.g. interpolation (Henzinger et al. 2004), unsat core extraction (Gupta and Strichman 2005) or weakest precondition (Podelski and Rybalchenko 2007)).

Policy Predicate Abstraction

In the previous section, we have revisited predicate abstraction for general safety verification. However, we consider safety under a policy π. Accordingly, we are not interested in the predicate abstraction of the full state space Θ but of the policy-restricted subgraph Θ^π. In this section, we revisit the policy predicate abstraction approach of Vea22. In the next section, we adapt CEGAR to enable policy verification via policy predicate abstraction.

Definition 3 (Policy Predicate Abstraction)**.** Let $\mathcal{P} \subseteq C$ be a predicate set, and let π be a policy. The **policy predicate abstraction** of Θ^π over $\mathcal{P}$ is the LTS $\Theta^\pi_{\mathcal{P}} = \langle \mathcal{S}_{\mathcal{P}}, \mathcal{L}, \mathcal{T}^\pi_{\mathcal{P}} \rangle$ where $\mathcal{T}^\pi_{\mathcal{P}} = \{(s|_{\mathcal{P}}, l, s'|_{\mathcal{P}}) \mid (s, l, s') \in \mathcal{T}, \pi(s) = l\}$.

In principle, the abstraction transition computation for $\Theta^\pi_{\mathcal{P}}$ can still be handled via calls to SMT. The new source of complexity is that one now additionally needs to check whether the policy π actually selects label l in $s \in [s_{\mathcal{P}}]$. Due to the non-linear structure of NN (specifically the activation functions) solving this abstract transition problem becomes computationally very expensive. Indeed, as observed by Vea22, naively querying general purpose SMT solvers is no longer feasible.

Algorithmic Enhancements by Vea22. To tackle this challenge, they devise a range of algorithmic enhancements based on relaxation and specialized NN analysis. Most notably, continuous-relaxation of the discrete integer state variables allows to query SMT solvers tailored to NN analysis (e.g. *Marabou* (Katz et al. 2017)). Vea22 then iterate relaxed transition problems in a branch & bound search; recursively branching over the state variables currently assigned to non-integer values by the relaxed solver. A branch is terminated once its relaxed (sub-)problem is found to be unsatisfiable,

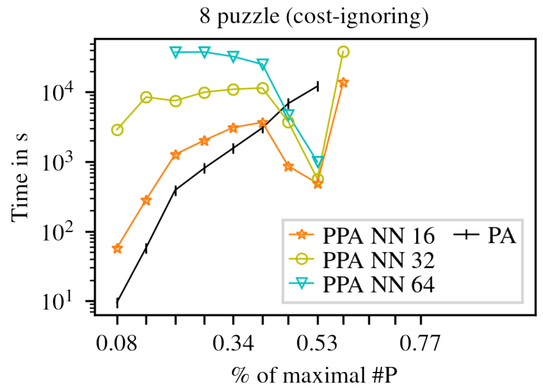

Figure 1: Result extracted from Vea22. PPA is the best-performing policy predicate abstraction variant (based on *Marabou* in branch & bound), and PA corresponds to standard predicate abstraction. The x-axis ranges over the number of predicates $|\mathcal{P}|$ in % of maximal $|\mathcal{P}|$.

or when the solution (found by the relaxed solver) becomes integer. If no integer solution is found during the search, then the exact problem in unsatisfiable.

Empirical Results. Vea22 found that their enhancements – especially the branch & bound approach – improve performance drastically. For their empirical evaluation, they used manually designed predicate sets $\mathcal{P}$ following a simple box constraint scheme $v \geq c$ with $v \in \mathcal{V}$ and $c \in D_v$. In their analysis they then examined performance as a function of the number of predicates $|\mathcal{P}|$. The positive results pertain to particular instances of $\mathcal{P}$, which are often located at sweetspots in the scaling scheme of $\mathcal{P}$.

Figure 1 shows an illustrative example of the results obtained by Vea22. The x-axis scales over $|\mathcal{P}|$. The y-axis shows the time to compute the fragment of the abstract state space reachable from ϕ_0, i.e., the fragment relevant for analyzing safety. One can clearly observe the sweetspot in the middle of the scaling scheme. “To the left”, for smaller and thereby coarser $\mathcal{P}$, the possible NN input regions of each abstract transition problem are significantly larger. Hence, the transition computation becomes drastically more expensive – especially for larger NN – and dominates the runtime to compute the abstract state space. This effect is reduced as $\mathcal{P}$ becomes finer. Additionally, for finer $\mathcal{P}$, we profit from the new gain in reachability reduction resulting from fixing the policy together with ϕ_0. In particular, for finer $\mathcal{P}$, policy predicate abstraction is less expensive than standard (non-policy-restricted) predicate abstraction. “To the right”, for growing $|\mathcal{P}|$, the state space explosion kicks in and eventually outweighs these effects. As a consequence, we obtain the sweet spot in the middle.

In summary, Vea22 leave us with two open questions: 1. how to find suitable $\mathcal{P}$ automatically, and 2. whether doing so incurs an infeasible overhead from computing abstract state spaces outside the sweetspot. The latter is of particular relevance in the context of CEGAR schemes, as here one usually starts with an initially coarse and thus in the policy-restricted context particularly expensive predicate set.

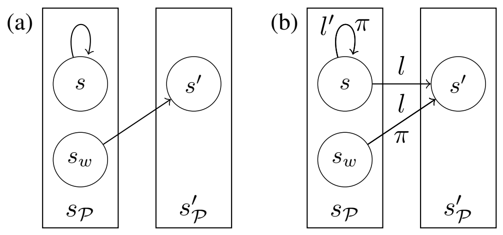

Figure 2: Illustration of standard spuriousness (a) and policy-related spuriousness (b), with s as the only concrete state in $s_{\mathcal{P}}$ reachable from ϕ_0 and s_w as the witness of the abstract transition $(s_{\mathcal{P}}, l, s'_{\mathcal{P}})$.

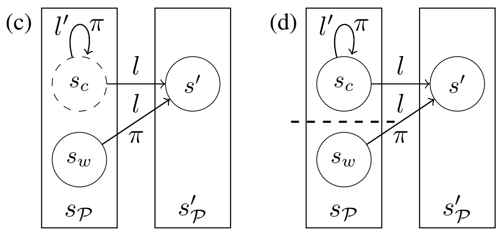

Figure 3: Illustration of policy-related refinement based on concretization exclusion (c) and witness splitting (d), with s_w as the witness of the abstract transition $(s_{\mathcal{P}}, l, s'_{\mathcal{P}})$ and s_c as the corresponding state in the concretization of the abstract path.

CEGAR

The safety result for standard predicate abstraction, i.e., in the non-policy-restricted case, translates to policy predicate abstraction in a straight forward manner.

Proposition 4 (Soundness)**.** Let $\rho = (\phi_0, \phi_u)$ be a safety property. If $\Theta^{\pi}_{\mathcal{P}}$ is safe with respect to ρ, then so is π.

Like for $\Theta_{\mathcal{P}}$, paths in $\Theta^{\pi}_{\mathcal{P}}$ may be spurious. However, compared to standard predicate abstraction, we can distinguish an additional source of spuriousness, namely *policy-related* spuriousness. Figure 2 illustrates the difference. In the standard case (a), the transition from $s_{\mathcal{P}}$ to $s'_{\mathcal{P}}$ is spurious due to the transitional behavior of the underlying system (in our case described by the operators $\mathcal{O}$). That is, while the abstract transition is witnessed by the concrete transition from $s_w \in [s_{\mathcal{P}}]$ to $s' \in [s'_{\mathcal{P}}]$, there does not exist a witness in the subset $\{s\}$ of $[s_{\mathcal{P}}]$, that is reachable from ϕ_0 in Θ (under the possible concretizations of some considered abstract path prefix). Now compared to (a), in the policy-related case (b), there may actually exist a reachable witness in $[s_{\mathcal{P}}]$, yet not under policy-restriction, i.e., $\pi(s) \neq l$.

In principle, refining $\mathcal{P}$ to remove policy-related spuriousness is still amenable to standard refinement techniques. $\pi(s) = l$ is essentially a highly complex transition guard, that, for instance, can be fed into weakest precondition computation. However, each computed "selection predicate" effectively adds an additional NN structure to the abstract transition problem. As observed by Vea22, solving such "multi NN" problems via SMT quickly becomes infeasible.[2]

Hence, we instead tackle policy-refinement via a more scalable approach that is based on the approximation of the policy selection condition via simple (linear) predicates. This has the decisive advantage, that the resulting abstract transition problems remain amenable to the algorithmic enhancements as already provided by Vea22. For the remainder of this section, we first describe two specific methods for policy-refinement via approximation. Afterwards, we combine these with a complete abstraction refinement algorithm based on weakest precondition computation.

[2]Strictly speaking, the SMT problems in Vea22 (used for their bounded-length verification baseline) are more complex, as, unlike for selection predicates, the NN structures are not constraint to a specific label. Still handling multi-NN SMT problems (as well as the selection predicates themselves) efficiently remains non-trivial (substantial implementation and engineering at least).

Policy-Related Refinement

Both of our methods for policy-related refinement assume a specific **concretization** $\langle s_c^0, o^0, \ldots, o^n, s_c^u \rangle$ of a corresponding (spurious) abstract path $\langle s^0_{\mathcal{P}}, o^0, \ldots, o^n, s^u_{\mathcal{P}} \rangle$. Figure 3 serves as illustration.

As a kind of naïve baseline **concretization exclusion** (c) removes the concretization state s_c, in which the policy selection $\pi(s_c) = l'$ disagrees with the corresponding abstract transition label l, as a singleton from the abstract block state $s_{\mathcal{P}}$. In other words, the selection condition is over-approximated as $\neg s_c$. In our current approach, we achieve this using the (more fine-grained) box constraint set $\{v \leq s_c(v) - 1, s_c(v) + 1 \leq v \mid v \in \mathcal{V}\}$.

As a more targeted method, **witness splitting** (d) additionally takes the witness s_w of the spurious abstract transition into account. It splits the abstract state into one half containing s_c and another containing s_w. The underlying idea is that the selection condition is approximated by the split containing s_w. The split can be achieved via any constraint ϕ such that $\phi(s_c) \neq \phi(s_w)$. In our current approach, we use box constraints of the form $s_w(v)\, \{\leq, \geq\}\, v$ for $s_c(v)\, \{<, >\}\, s_w(v)$ (see procedure `WS` line 23 in Algorithm 1 for details). Future work will investigate how to select linear constraints different from box constraints but also how to optimize the state variable subset used for the split.

Abstraction Refinement

Algorithm 1 shows the pseudo-code for our abstraction refinement. Given an abstract unsafe path $\langle s^0_{\mathcal{P}}, o^0, \ldots, o^n, s^u_{\mathcal{P}} \rangle$ (found by search in $\Theta^{\pi}_{\mathcal{P}}$) we check spuriousness and refine with respect to the operator path $\langle o^0, \ldots, o^n \rangle$ and abstract start state $s^0_{\mathcal{P}}$. That is, while our method ignores spuriousness with respect to the abstract state postfix $\langle s^1_{\mathcal{P}}, \ldots, s^u_{\mathcal{P}} \rangle$, $s^0_{\mathcal{P}}$ is necessary to guarantee completeness (see below).

Algorithm 1 starts by checking standard spuriousness, i.e., without the policy. It incrementally checks whether there exists a start state (grouped to the abstract start state $s^0_{\mathcal{P}}$) such that a prefix of the operator path can be taken (line 2 et sqq). Similar to the abstract transition computation such path existence checks can be encoded as SMT queries. If some prefix cannot be concretized, we refine with respect to this (smallest) spurious prefix (line 3). If the complete path can be concretized, Algorithm 1 checks whether there exists

Algorithm 1: Abstraction Refinement.

```
Input: $s^0_\mathcal{P} \models \phi_0$ and $\langle o^0, \ldots, o^n \rangle$ with $o^i = (g^i, l^i, u^i)$.
   // Check standard spuriousness.
1  for $i \in \{0, \ldots, n\}$ do
2      if $\neg\exists\langle s^0, o^0, \ldots, s^i, o^i \rangle \in \Theta : s^0 \in [s^0_\mathcal{P}] \wedge s^0 \models \phi_0 \wedge s^i \models g^i$ then
3          $\mathcal{P} \leftarrow \mathcal{P} \cup$ WP ($g^i$, $\langle o^0, \ldots, o^{i-1} \rangle$ )
4          return SPURIOUS
5  if $\neg\exists\langle s^0, o^0, \ldots, o^n, s^u \rangle \in \Theta : s^0 \in [s^0_\mathcal{P}] \wedge s^0 \models \phi_0 \wedge s^u \models \phi_u$ then
6      $\mathcal{P} \leftarrow \mathcal{P} \cup$ WP ($\phi_u$, $\langle o^0, \ldots, o^n \rangle$ )
7      return SPURIOUS
   // Check $\pi$-spuriousness.
8  let $\langle s^0_c, o^0, \ldots, o^n, s^u_c \rangle \in \Theta : s^0_c \in [s^0_\mathcal{P}] \wedge s^0_c \models \phi_0 \wedge s^u_c \models \phi_u$ in
9      for $i \in \{0, \ldots, n\}$ do
10         if $\pi(s^i_c) \neq l^i$ then
11             if $\pi$-Refinement $= WS$ then
12                 $\phi_{appr} \leftarrow$ WS ($s^i_c, s^i_{witness}$)
13             else
14                 $\phi_{appr} \leftarrow \neg s^i_c$
15             $\mathcal{P} \leftarrow \mathcal{P} \cup$ WP ($\phi_{appr}$, $\langle o^0, \ldots, o^{i-1} \rangle$ )
16             return $\pi$-SPURIOUS for $s^0$
17 return NON-SPURIOUS
18 Procedure WP ($\phi, \langle o^0, \ldots, o^i \rangle$):
19     $\phi^{i+1}_{wp} \leftarrow \phi$
20     for $j \in \{i, \ldots, 0\}$ do
21         $\phi^j_{wp} \leftarrow wp_{u^j}(\phi^{j+1}_{wp})$
22     return $\{\phi^0_{wp}, \ldots, \phi^i_{wp}\}$
23 Procedure WS ($s_c, s_w$):
24     $\phi \leftarrow \{\}$
25     for $v \in \mathcal{V}$ s.t. $s_c(v) \neq s_w(v)$ do
26         if $s_c(v) < s_w(v)$ then $\phi \leftarrow \phi \cup \{s_w(v) \leq v\}$
27         if $s_c(v) > s_w(v)$ then $\phi \leftarrow \phi \cup \{s_w(v) \geq v\}$
28     return $\phi_{wp}$
```

a concretization such that additionally the target state s^u is unsafe (line 5 et sqq). Again, if no, we refine (line 6).

If the path is non-spurious without the policy, Algorithm 1 proceeds by checking policy-related spuriousness. It picks a specific concretization path in Θ, i.e., a path already valid up to π (line 8), and then incrementally checks whether the path is also valid under π. If no (line 10 et sqq), the path is *π-spurious for s^0_c* and we refine either according to witness splitting (line 12) or according to concretization exclusion (line 14) and with respect to the spurious prefix (line 15).[3] If yes (line 17), we have found a concrete unsafe path, thus proving unsafety.

Given an operator prefix $\sigma = \langle o^0, \ldots, o^i \rangle$ and a constraint ϕ, procedure WP (line 18) describes the weakest precondition computation that is used to refine standard spuriousness (lines 3 & 6) as well as policy-related spuriousness (line 15). Here, $wp_u(\phi^j)$, denotes the syntactical weakest precondition for ϕ^j applying update u, which is computed by substituting each $v \in dom(u)$ in ϕ^j by $u(v)$. In essence, WP computes the set $\{\phi^0_{wp}, \ldots, \phi^{i+1}_{wp}\}$ where ϕ^j_{wp} is the weakest precondition for ϕ taking $\langle o^j, \ldots, o^i \rangle$. Intuitively, this enables the refined abstraction to explicitly trace the truth value of ϕ taking σ. For the remainder of this section, we investigate formally how this achieves completeness.

Proposition 5 (Standard). Let $\{\phi^0_{wp}, \ldots, \phi^{i+1}_{wp}\} \subseteq \mathcal{P}$ as computed by $\texttt{WP}(\phi, \langle o^0, \ldots, o^i \rangle)$, i.e., $\phi^{i+1}_{wp} = \phi$: For any $\langle s^0_\mathcal{P}, o^0, \ldots, o^i, s^{i+1}_\mathcal{P} \rangle$ in $\Theta_\mathcal{P}$ with $s^{i+1}_\mathcal{P} \models \phi^{i+1}_{wp}$, if $\langle s^0, o^0, \ldots, o^i, s^{i+1} \rangle$ in Θ with $s^0 \in [s^0_\mathcal{P}]$, then also $s^{i+1} \models \phi^{i+1}_{wp}$.

Proof. Since $s^{i+1}_\mathcal{P} \models \phi^{i+1}_{wp}$, we have $s^{i+1}_\mathcal{P}(\phi^{i+1}_{wp}) = 1$ and thus, by weakest precondition, also $s^0_\mathcal{P}(\phi^0_{wp}) = 1$. Hence, since $s^0 \in [s^0_\mathcal{P}]$, $s^0 \models \phi^0_{wp}$, and therefore, again by weakest precondition, $s^{i+1} \models \phi^{i+1}_{wp}$. □

In essence, Proposition 5 tells us, that if there still exists a (spurious) path prefix $\sigma = \langle s^0_\mathcal{P}, o^0, \ldots, o^i, s^{i+1}_\mathcal{P} \rangle$ such that $s^{i+1}_\mathcal{P} \models \phi^{i+1}_{wp}$ after the refinement step, then it is spurious due to a prefix of σ. This prefix can be detected in subsequent iterations. Since in each iteration, the size of the spurious prefix is "at least" strictly decreased, it is removed completely within finitely many steps.[4] Moreover, since the guarantees provided by Proposition 5 are bound to concretization paths from $s^0_\mathcal{P}$, we now can also see that constraining the incremental path existence check in Algorithm 1 by $s^0_\mathcal{P}$ is actually necessary to guarantee progress.

Proposition 6 (Concretization Exclusion). Let $\{\phi^0_{wp}, \ldots, \phi^{i+1}_{wp}\} \subseteq \mathcal{P}$ as computed by $\texttt{WP}(s_c, \langle o^0, \ldots, o^i \rangle)$, with $\pi(s_c) \neq l$: For any $\langle s^0_\mathcal{P}, o^0, \ldots, o^i, s^{i+1}_\mathcal{P} \rangle$ in $\Theta^\pi_\mathcal{P}$ with $l \in \pi(s^{i+1}_\mathcal{P})$, and any $\langle s^0_c, o^0, \ldots, o^i, s_c \rangle$ in Θ, it holds $s^0_c \notin [s^0_\mathcal{P}]$.

Proof. Since $l \in \pi(s^{i+1}_\mathcal{P})$, while $s_c|_\mathcal{P} = \{s_c\}$ and $\pi(s_c) \neq l$, we have $s_c \notin [s^{i+1}_\mathcal{P}]$ and thus $s_\mathcal{P}(\phi^{i+1}_{wp}) = 1$ (where $\phi^{i+1}_{wp} = \neg s_c$). Hence, by weakest precondition, $s^0_\mathcal{P}(\phi^0_{wp}) = 1$. Moreover, since, again by weakest precondition, $s^0_c \not\models \phi^0_{wp}$, it follows $s^0_c \notin [s^0_\mathcal{P}]$. □

Proposition 6 formalizes the intuition behind concretization exclusion, i.e., s_c (respectively the path to s_c) is excluded from the set of concretizations – again, this only holds for path existence constrained to the abstract start state

[3]Since we only consider a specific concretization, a path π-spurious for s^0_c may still be non-spurious for some other $s'_c \in [s^0_\mathcal{P}]$. However, our π-refinement methods assume a specific concretization. Moreover, as argued before, handling mulit-NN SMT problems (in the form of path existence) efficiently is non-trivial and remains future work.

[4]A more aggressive refinement approach would also include the prefix's guards during the weakest precondition computation; then removing the spurious prefix within a single iteration. Preliminary experiments indicate that this adds to many non-relevant predicates, thereby decreasing performance significantly.

$s_{\mathcal{P}}^0$. Since in each iteration we remove "at least" one state 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.

Proposition 7 (Witness Splitting)**.** Let $\{\phi_{wp}^0, \dots, \phi_{wp}^{i+1}\} \subseteq \mathcal{P}$ as computed by $\texttt{WP}(\texttt{WS}(s_c, s_w), \langle o^0, \dots, o^i \rangle)$, For any path $\langle s_{\mathcal{P}}^0, o^0, \dots, o^i, s_{\mathcal{P}}^{i+1} \rangle$ in $\Theta_{\mathcal{P}}^{\pi}$ such that there exists $s_c^0 \in [s_{\mathcal{P}}^0]$ with $\langle s_c^0, o^0, \dots, o^i, s_c \rangle$ in Θ, it holds $s_w \notin [s_{\mathcal{P}}^{i+1}]$.

Proof. By weakest precondition, from $s_c \not\models \phi_{wp}^{j+1}$ (where $\phi_{wp}^{i+1} = \texttt{WS}(s_c, s_w)$) it follows $s_c^0 \not\models \phi_{wp}^0$. Hence, if $s_c^0 \in [s_{\mathcal{P}}^0]$ then $s_{\mathcal{P}}^0(\phi_{wp}^0) = 0$. Thus, again by weakest precondition, $s_{\mathcal{P}}^{i+1}(\phi_{wp}^{i+1}) = 0$, and, since $s_w \models \phi_{wp}^{i+1}$, we thereby have $s_w \notin [s_{\mathcal{P}}]$. □

Proposition 7 captures the idea behind witness splitting. That is, while there still may be an abstract path with the path to s_c as a possible concretization (prefix), such an abstract path may no longer contain s_w as a witness to the transition from $s_{\mathcal{P}}^{i+1}$. Again, also here, constraining the abstract start state is actually necessary. Since in each iteration we remove "at least" one witness from the abstract state to which s_c is grouped, the path to s_c is excluded as a concretization candidate within finitely man steps.

In summary, a refinement loop using Algorithm 1 removes in each iteration not necessarily "all" but "at least some" spuriousness; hence guaranteeing completeness of the verification approach:

Proposition 8 (Completeness)**.** Starting with an arbitrary predicate set $\mathcal{P}$, an iterative search for a path from ϕ_0 to ϕ_u in $\Theta_{\mathcal{P}}^{\pi}$ with refinement as per Algorithm 1 will in finitely many iterations either find such a path that is non-spurious, or terminate with refined $\mathcal{P}$ for which such a path does not exist.

Optimizations

Independent of the specific method used for abstraction refinement, CEGAR allows for various optimizations to improve on the abstract state space computation itself. In our current work, we explore two of these.

Heuristic Search. In the context of our abstraction refinement loop, there is no need to re-compute the complete abstract state space $\Theta_{\mathcal{P}}^{\pi}$ in each iteration. Instead, we can initiate an abstract search from ϕ_0 (i.e., the set of abstract start states $\{s_{\mathcal{P}}^0 \in \mathcal{S}_{\mathcal{P}} \mid s_{\mathcal{P}}^0 \models \phi_0\}$) and stop as soon as an unsafe path has been found. The complete fragment (reachable from $s_{\mathcal{P}}^0$) is then computed only for the final predicate set (and only if π is actually safe).

Following this standard idea, we want to improve even further. In the spirit of AI planning, we run a heuristic search. That is, given a heuristic h estimating the distance to an abstract unsafe state, we expand (reached) abstract states greedily according to the smallest distance estimated by h. While the idea seems quite natural, we are not aware of prior work using heuristic search in a CEGAR context.

In our work so far, we use a simple hamming distance heuristic:

Definition 9 (Hamming Distance)**.** Let $\mathcal{P}_u^0 = \{p \in \mathcal{P} \mid \phi_u \vdash \neg p\}$ and $\mathcal{P}_u^1 = \{p \in \mathcal{P} \mid \phi_u \vdash p\}$. The **hamming distance** of abstract state $s_{\mathcal{P}} \in \mathcal{S}_{\mathcal{P}}$ is $h_d(s_{\mathcal{P}}) = \left|\{p \in \mathcal{P}_u^0 \mid s_{\mathcal{P}}(p)\}\right| + \left|\{p \in \mathcal{P}_u^1 \mid \neg s_{\mathcal{P}}(p)\}\right|$.

Intuitively, h_d is similar to the goal counting heuristic known from AI planning. It counts the predicates for which $s_{\mathcal{P}}$ differs from the truth value entailed by the unsafety condition ϕ_u, $\mathcal{P}_u^0$ are the predicates entailed to be false and $\mathcal{P}_u^1$ are the predicate entailed to be true. As our experiments show, already this simple heuristic often improves performance drastically.

Incremental Computation of $\Theta_{\mathcal{P}}^{\pi}$. Another optimization, that we deploy, is the incremental computation of the abstract state space $\Theta_{\mathcal{P}}^{\pi}$. Similar optimizations have already been utilized in other CEGAR contexts (e.g., (Henzinger et al. 2002)). Intuitively, rather than starting from scratch, we can reuse the transition information of already computed coarser abstractions when computing the refined abstract state space. Proposition 10 states the underlying preservation properties:

Proposition 10. Given predicate sets $\mathcal{P} \subseteq \mathcal{P}'$ and abstract states $s_{\mathcal{P}}^1 \subseteq s_{\mathcal{P}'}^1, s_{\mathcal{P}}^2 \subseteq s_{\mathcal{P}'}^2$ then:

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

By construction, $\mathcal{T}_{\mathcal{P}}^{\pi}$ over-approximates $\mathcal{T}_{\mathcal{P}'}^{\pi}$. Hence, as captured by (1.), we can immediately skip all transition problems that correspond to transitions not possible under coarser abstractions. For instance, given $\mathcal{P} = \{p\}$ and $\mathcal{P}' = \{p, p'\}$. If $(\{p \mapsto 1\}, l, \{p \mapsto 0\}) \notin \mathcal{T}_{\mathcal{P}}^{\pi}$ then also $(\{p \mapsto 1, p' \mapsto 0\}, l, \{p \mapsto 0, p' \mapsto 1\}) \notin \mathcal{T}_{\mathcal{P}'}^{\pi}$. Conversely, as captured by (2.), we can also reuse any transition witness already during the computation of coarser abstractions. Crucially, the (source) witness s_w as well as its successor s' have to be in the concretization of the abstract source respectively abstract successor state in the refined abstraction.

Experiments

Experiments Setup

For our empirical evaluation, we extended the implementation of Vea22 to support our CEGAR approach. Throughout our experiments we use their best-performing configuration for abstract transition computation (based on querying *Marabou* (Katz et al. 2019) in branch & bound for continuously-relaxed transition problems). For all other SMT queries we use $Z3$ (de Moura and Bjørner 2008). All experiments were run on machines with Intel Xenon E5-2650 processors with a clock rate of 2.2 GHz, with time and memory limits of 12 h and 4 GB respectively.

Algorithmic Configurations. We evaluated different configuration of our CEGAR approach. **WS** uses witness splitting to refine policy-related spuriousness and applies all optimizations (heuristic search and incremental computation). **~~h~~**$_{\cancel{d}}$ disables the heuristic search, while **~~inc~~** disables the incremental computation. Alternatively, **CE** uses concretiaztion-exclusion to refine policy-related spuriousness (with all op-

Benchmark	NN	Time in s	$\%P_{max}$	# It.	π-Ref.	Safe	BMC
		WS|~~h_d~~|~~inc~~|CE	WS|~~h_d~~|~~inc~~|CE	WS|~~h_d~~|~~inc~~|CE	WS|~~h_d~~|~~inc~~|CE		
	NN 16	8.6|12.6|10.9|**4.1**	10|10|11|10	10|10|10|6	3|3|3|1	✓	-
4 Blocks	NN 32	13.4|22.3|14.2|**6.9**	11|11|11|9	9|9|9|5	3|3|3|1	✓	-
(cost-ignoring)	NN 64	81.9|310.4|112.5|**53.5**	11|11|10|10	10|10|9|7	2|2|2|1	✓	-
	NN 16	**255.3**|959.5|302.7|360.0	33|31|33|29	23|20|23|19	4|4|4|1	✓	-
6 Blocks	NN 32	280.7|3659.5|337.2|**208.6**	31|31|31|27	18|17|18|17	5|5|5|2	✓	-
(cost-ignoring)	NN 64	5934.2| - |8929.9|**1147.8**	28|10|28|25	19|4|19|12	7|1|7|1	✓	-
	NN 16	29155.7| - |**20036.0**|31234.4	44|41|46|43	32|28|38|40	7|7|6|1	✓	-
8 Blocks	NN 32	- | - | - | -	43|28|42|40	36|14|34|36	4|4|4|1	-	-
(cost-ignoring)	NN 64	- | - | - | -	10|8|10|22	5|4|5|10	1|1|1|1	-	-
	NN 16	280.5|3936.3|300.9|221.5	36|40|36|31	26|31|26|13	18|21|18|9	×	**73.9**
8 Puzzle	NN 32	39868.1| - |42098.4|**31168.1**	90|45|90|95	72|28|72|58	14|13|14|7	✓	-
(cost-ignoring)	NN 64	**42285.7**| - |42813.3| -	93|42|93|72	66|30|66|38	12|14|12|7	✓	-
	NN 16	370.4|442.6|378.8|**88.5**	14|14|14|11	11|11|11|7	4|4|4|1	✓	-
4 Blocks	NN 32	604.2|827.5|5041.1|**599.1**	14|15|23|12	11|12|18|7	5|5|9|1	✓	-
(cost-aware)	NN 64	**25081.5**| - |25893.1| -	16|15|16|12	10|9|10|6	4|4|4|1	✓	-
	NN 16	**2413.2**| - |2524.1|27314.0	35|31|35|37	21|17|21|23	6|6|6|2	✓	-
6 Blocks	NN 32	**1323.7**|8650.9|1344.6| -	35|36|35|28	17|18|17|12	5|5|5|1	✓	-
(cost-aware)	NN 64	- | - | - | -	21|15|16|26	6|4|5|8	2|1|1|2	-	-
	NN 16	169.2|28756.8|207.5|**10.0**	43|43|43|25	13|13|13|4	8|8|8|1	×	61.9
8 Blocks	NN 32	- | - | - | -	43|32|43|28	20|10|20|9	8|5|8|1	-	-
(cost-aware)	NN 64	- | - | - | -	14|14|14|27	4|4|4|7	1|1|1|2	-	-
	NN 16	8039.2| - |9342.9|**7383.4**	64|61|64|41	30|27|30|24	15|15|15|8	×	32965.7
8 Puzzle	NN 32	- | - | - | -	63|59|61|53	34|25|33|38	13|14|13|5	-	-
(cost-aware)	NN 64	- | - | - | -	58|10|58|52	25|5|25|21	11|1|11|9	-	-
	NN 16	1.3|**1.2**|**1.2**|335.4	9|9|9|20	4|4|4|6	3|3|3|5	×	57.0
Transport	NN 32	**63.7**|68.0|67.4|360.7	22|22|22|20	22|22|22|8	20|20|20|7	×	20487.1
	NN 64	**1.4**|**1.4**|**1.4**|608.6	7|7|7|22	3|3|3|9	2|2|2|8	×	-

Table 1: Results for the evaluated CEGAR configurations and the BMC competitor over different benchmarks and NN policies (distinguishing cost-aware policies and cost-ignoring policies where applicable.) For CEGAR we compare runtime, % of final $|\mathcal{P}|$ relative to maximal $|\mathcal{P}|$, the number of cegar iterations and the number of iterations with π-related refinement. - indicates timeouts (exceeding the 12h time limit).

timizations enabled). All configurations are initialized with the **empty predicate** set $\mathcal{P} = \emptyset$.

Competing Approach. As a competitor we picked up the *bounded model checking* (BMC) implementation of Vea22, which encodes bounded-length path existence from ϕ_0 to ϕ_u into SMT queries, incrementally increasing the path length L until an unsafe path is found. To enable BMC to also prove safety, we combine it with k-induction (Sheeran, Singh, and Stålmarck 2000). That is, additionally to the unsafety check, we also incrementally check the existence of a loop-free path of length L via SMT queries. The non-existence of such a path constitutes a sufficient condition for reaching a fix-point in the set of states reachable via paths up to length L. In other words, if there is does not exist an unsafe state reachable within $L-1$ steps, then ϕ_u is unreachable and safety is proven. To the advantage of the BMC competitor, we run the fix-point and unsafety check independently of each other; reporting the runtime of first successful check as the one of BMC. In fact, on the considered benchmarks, BMC never reaches a path length at which the fix-point check succeeds. In other words it never achieves to prove safety.

Benchmarks. We experimented with non-deterministic variants of the planning domains Blocksworld, SlidingTiles and Transport as introduced by Vea22. In Blocksworld and SlidingTiles, actions moving a block/tile x may non-deterministically fail, and when this happens the cost of moving x (represented by an additional state variable) is incremented. 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. Here, we use an 8-puzzle instance. In Transport, a truck must deliver packages on a straight-line road; (safely) crossing a bridge with limited capacity.

We also reused the policies of Vea22, i.e., for every considered domain instance three feed-forward NN policies trained by Q-learning (Mnih et al. 2015), each with 2 hidden layers and 16, 32, respectively 64 neurons per layer. Like Vea22, on Blocksworld and SlidingTiles, we considered policies hat do vs. do not take move costs into account.

On the considered planning domains, the refinement predicates added by Algorithm 1 are all of the form $v \geq c$ where $v \in \mathcal{V}$ and $c \in D_v$. We hence can compare the computed predicate sets to the predicate set of maximal size, i.e., where all variable values can be distinguished, cf., Table 1.

Experiments Results

We now discuss the results of our current evaluation. An overview is provided in Table 1

Comparison of Policy-Related Refinement. Comparing CE and WS, none clearly outperforms the other. WS has a larger coverage (20 out of 27 problem instance compared to 17 by CE). Conversely, in terms of runtime, CE beats WS more often (11 problem instance compared to 9 by WS).

Moreover, CE usually requires fewer policy-related refinements and fewer iterations in general. If so, it typically also collects fewer predicates – even tough not necessarily (see, e.g., 4 Blocks and NN 16 cost-ignoring). These observations are to be expected. Since, for policy-related refinement, we add the box constraint split of $\neg s_c$, CE pursues a rather aggressive refinement strategy, possibly adding many predicates not actually needed. However, on smaller instances, where such a strategy is still feasible, CE quickly converges to a predicate set sufficiently fine grained to decided safety. Indeed, CE wins on 5 out of 6 of the rather small 4 Blocks instances. On larger instances, the more restrained witness splitting method dominates.

Optimizations. The heuristic search optimization improves performance drastically. In particular, ~~h_d~~, i.e., the configuration with heuristic search disabled, fails to terminate on 7 instances covered by WS. As the only exception, ~~h_d~~ is able to compete on the transport instances. However, this is due to the problem structure of the transport domain. Here, ϕ_u is composed of a single flag that is set when crossing the bridge too heavily loaded. As a consequence, the hamming distance degenerates to a boolean check for $s_{\mathcal{P}} \models \phi_u$, and is thus completely useless. Clearly, this is heuristic dependent and can be expected to change for other/more advanced heuristics.

While incremental computation also tends to improve performance, its impact seems to be less drastic. ~~inc~~ covers all instances covered by WS. Moreover, there is actually one instance on which ~~inc~~ dominates WS. This is possible as incremental computation affects the used transition witnesses and thereby the witness splitting. Indeed, on the instance dominated by ~~inc~~, it also requires fewer π-refinements. Still, usually the impact of both optimizations to the predicate selections seem to be rather limited.

BMC and Safety. Comparing our CEGAR approach to BMC, the latter is usually outperformed by the former. There is one instance where BMC is faster than both (WS and CE), and two other instances where it is either faster than WS or CE. It fails to find an unsafe path on 1 out of the 6 problem instances proved unsafe by CEGAR and is, in parts, drastically outperformed (see, e.g., transport NN 32). Moreover, BMC never achieves to prove safety. A major bottleneck is here the absence of NN tailored techniques – as available for abstract transition problems – that are applicable to the path existence problems faced by BMC.[5]

Many of the policies considered in our evaluation are proven unsafe (6 out of 20 covered instances). Following the ideas of Vea22, we note that our verification approach can, in principle, be adapted to compute per-(abstract)-state safety. That is, even if the policy is unsafe for a single start state, there may still be many start states for which no unsafe path exists. In the context of our CEGAR approach this means that, even if we have found a non-spurious unsafe path for some abstract start state $s^0_{\mathcal{P}}$ thus proving unsafety (say with concretization $s^0_c \in [s^0_{\mathcal{P}}]$), we may continue to verify safety for the remaining (abstract) start states ($\phi_0 \setminus \{s^0_c\}$ or more coarsely $\phi_0 \setminus [s^0_{\mathcal{P}}]$). This is another advantage over BMC, which, conceptually, can only be used to prove or disprove unsafe path existence.

[5](Amir, Schapira, and Katz 2021) use *Marabou* for BMC. This is however in a NNCS setting, i.e., providing the NN output directly as input to a continuous state evolution function. In constrast, in the context of multi-NN problems, *Marabou* provides no native support for label selection as required for NN action policies.

Conclusion

The verification of neural network behavior becomes more and more important. We have extended the recent technique of policy predicate abstraction to support fully automatic verification via a CEGAR approach based on the approximation of policy selection conditions. Our empirical results show that the approach is feasible, thanks in particular to the usage of heuristic search during the abstract state space exploration, and can outperform other methods.

Future work will continue to investigate the methods introduced for policy-related refinement, e.g., fewer predicates per refinement and predicates different from box constraints. We might also explore the adaption to NN policies of other refinement methods in literature (e.g. (Henzinger et al. 2004; Gupta and Strichman 2005)).

Furthermore, we plan to use standard predicate abstraction to compute heuristics that improve on the hamming distance used so far. Alternatively, one may try to reuse information obtained from coarser (policy-restricted) abstractions to guide the search in the refined state space. We may also consider lazy abstraction (Henzinger et al. 2002), i.e., to only refine the abstraction in a region local to the detected spuriousness. Additionally, one may experiment with "on-demand" policy-restriction, i.e., to only compute policy predicate abstraction on abstract unsafe path found under standard abstraction.

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 *KR*.

Akintunde, M. E.; Kevorchian, A.; Lomuscio, A.; and Pirovano, E. 2019. Verification of RNN-Based Neural Agent-Environment Systems. In *AAAI*.

Amir, G.; Schapira, M.; and Katz, G. 2021. Towards Scalable Verification of Deep Reinforcement Learning. In *Formal Methods in Computer Aided Design, FMCAD*.

Ball, T.; Majumdar, R.; Millstein, T. D.; and Rajamani, S. K. 2001. Automatic Predicate Abstraction of C Programs. In *Prog. Lang. Design and Implementation (PLDI)*.

Barrett, C. W.; Sebastiani, R.; Seshia, S. A.; and Tinelli, C. 1994. Satisfiability modulo theories. *In Handbook of Satisfiability*, 825–885.

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

Clarke, E. M.; Grumberg, O.; Jha, S.; Lu, Y.; and Veith, H. 2000. Counterexample-Guided Abstraction Refinement. In *CAV*.

de Moura, L.; and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In *TACAS*.

Dutta, S.; Chen, X.; and Sankaranarayanan, S. 2019. Reachability analysis for neural feedback systems using regressive polynomial rule inference. In *Intl Conf Hybrid Systems: Computation and Control (HSCC)*.

Edelkamp, S. 2001. Planning with Pattern Databases. In *ECP*.

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

Graf, S.; and Saïdi, H. 1997. Construction of Abstract State Graphs with PVS. In *CAV*.

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

Gupta, A.; and Strichman, O. 2005. Abstraction Refinement for Bounded Model Checking. In Etessami, K.; and Rajamani, S. K., eds., *CAV*.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge & Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *JACM*, 61(3): 16:1–16:63.

Henzinger, T. A.; Jhala, R.; Majumdar, R.; and McMillan, K. L. 2004. Abstractions from proofs. In *Principles of Programming Languages (POPL)*.

Henzinger, T. A.; Jhala, R.; Majumdar, R.; and Sutre, G. 2002. Lazy abstraction. In Launchbury, J.; and Mitchell, J. C., eds., *Principles of Programming Languages (POPL)*.

Huang, S.; Fan, J.; Li, W.; Chen, X.; and Zhu, Q. 2019. ReachNN: Reachability analysis of neural-network controlled systems. *ACM Trans. Emb. Comp. Sys.*, 18: 1–22.

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

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. Emb. Comp. Sys.*, 20: 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 *CAV*.

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 *CAV*.

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

Seipp, J.; and Helmert, M. 2018. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *JAIR*, 62: 535–577.

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*.

Sun, X.; Khedr, H.; and Shoukry, Y. 2019. Formal Verification of Neural Network Controlled Autonomous Systems. In *Intl Conf Hybrid Systems: Computation and Control (HSCC)*.

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 *ICAPS*.