The Thirty-Ninth AAAI Conference on Artificial Intelligence (AAAI-25)

ASP-Driven Emergency Planning for Norm Violations in Reinforcement Learning

Sebastian Adam, Thomas Eiter

Institute of Logic and Computation, Vienna University of Technology
Favoritenstraße 9-11, A-1040 Vienna, Austria
{*sebastian.adam, thomas.eiter*}@tuwien.ac.at

Abstract

Reinforcement learning (RL) is a widely used approach for training an agent to maximize rewards in a given environment. Action policies learned with this technique see a broad range of applications in practical areas like games, healthcare, robotics, or autonomous driving. However, enforcing ethical behavior or norms based on deontic constraints that the agent should adhere to during policy execution remains a complex challenge. Especially constraints that emerge after the training can necessitate to redo policy learning, which can be costly and, more critically, time consuming. To mitigate this problem, we present a framework for policy fixing in case of a norm violation, which allows the agent to stay operational. Based on answer set programming (ASP), emergency plans are generated that exclude or minimize cost of norm violations by future actions in a horizon of interest. By combining and developing optimization techniques, efficient policy fixing under real-time constraints can be achieved.

Git — https://github.com/S3basuchian/emergency-planning

Introduction

Reinforcement learning (RL) is a well-known approach to train an agent for making decisions on actions in order to maximize cumulative rewards. Action policies obtained by RL proved to be effective in many areas, such as games (Silver et al. 2018), robotics (Kober, Bagnell, and Peters 2013), healthcare (Yu et al. 2023), or autonomous driving (Kiran et al. 2022), to name a few.

However, enforcing ethical behavior or norms based on deontic constraints that a trained agent should adhere to in operation is a complex challenge. It has garnered significant research attention in research related to RL (Thananjeyan et al. 2021; Noothigattu et al. 2018), deontic logic (Neufeld et al. 2022; Giordano, Martelli, and Dupré 2013) and planning (Kasenberg and Scheutz 2018; Aminof et al. 2019). Especially norms and constraints that emerge only after the training phase are difficult to factor into the agent's behavior.

For example, consider an RL agent in a high-stake area like autonomous driving. Even if the agent is trained to follow all road traffic rules, it could still overtake cyclists at an unsettling velocity. While not against the traffic rules per se, enforcing a norm that obliges the agent to overtake cyclists only if doable safely and at a reasonable velocity is desirable.

In traditional RL, incorporating such an additional norm into the agent's behavior is inevitably tied to redo training. However, training for a comprehensive policy can be expensive and, more critically, time-consuming, making it infeasible during operation. Some approaches (Thananjeyan et al. 2021; Noothigattu et al. 2018) thus outsource norm compliance to a secondary policy, while Neufeld et al. (2022) conceived a logic component that checks any candidate action against a set of norms and hinders the agent from execution in case of a norm violation. However, it lacks to account for norm violations by future actions. A hybrid approach by Kasenberg and Scheutz (2018) combines norms expressed in Linear Temporal Logic (LTL) with stochastic domains modeled by Markov decision processes (MDPs), but is geared to generate a norm-compliant policy from scratch rather than integrating it with an existing RL policy.

For handling norm violations, we propose to use emergency action plans that are generated based on a domain model. They ensure that no further norm violations happen and at the same time should closely follow the learned policy. In this way, the agent stays operational and intuitively still may achieve high reward while behaving norm compliant. More specifically, we make the following contributions:

- We introduce the notion of a policy fix, which consists in an alteration of the RL policy such that subsequent actions do not violate deontic constraints. To this end, we present a framework for policy fixing within an action horizon into the future that is gradually extended. Informally, a fix alters decisions to suboptimal choices while respecting action preferences by the RL policy.
- We express policy fixing as a planning problem in Answer Set Programming (ASP), which allows to compute optimal policy fixes using an ASP solver by respecting RL preferences. Norm compliance is thus entirely outsourced to an ASP program. If full norm compliance is impossible, ASP allows to assign penalties to norm violations using weak (soft) constraints. This facilitates sophisticated policy fixes, optimized based on expected rewards and penalties.
- The framework caters for both deterministic and nondeterministic environments, which are complex to handle. To do so, we introduce a blueprint for the ASP program that can be adapted to various complex domains, following a modular

Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

composition. Furthermore, we describe an innovative combination of optimization methods for program evaluation to achieve scalability to large problem instances.

Our experiments using the sota ASP solver *clingo* show that policy fixes work well in various scenarios, including ones that require fast response times. The framework we propose is thus well-equipped to serve as an interim solution until a comprehensive RL policy incorporating newly emerged norms is trained, which is crucial in scenarios like games or autonomous driving. Furthermore, the ASP approach is flexible with respect to model elaboration and amenable to property analysis and transparent for explainability.

Preliminaries

State transition system. A transition system describes and represents the dynamic behavior of an environment. It is formally defined as a tuple $T = (S, A, R, s_0)$, where:

- S is a set of states. Each state $s \in S$ represents a possible configuration of the environment at any point in time.
- A is a set of actions. An action $a \in A$ is an input or occurrence that causes transitions between states. Actions can be initiated by an agent.
- R is a mapping $S \times A \rightarrow \mathcal{P}(S)$ where $\mathcal{P}(S)$ is the power set (the set of all subsets) of S.
- $s_0 \in S$ is the initial state.

Such a transition system allows for multiple successor states to each state-action combination and is nondeterministic in general. The deterministic variant is the special case in which $|R(s, a)| \leq 1$ for each $s \in S$ and $a \in A$, i.e., every state-action combination has at most one successor state.

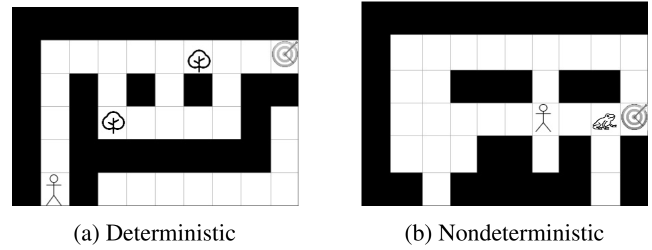
In dynamic environments, evolution over time is of particular concern. This is reflected by the definition of *trajectories* in a transition system, which are sequences $\sigma = s_0 a_1 ... a_n s_n$ of alternating states and actions where $s_{i+1} \in R(s_i, a_{i+1})$ for all $i \in [0, n)$. Hence, a given starting state and a sequence of actions can lead to multiple trajectories.

A plan in a deterministic environment is an action sequence $\alpha_n = a_1 ... a_n$, for which the resulting trajectory $s_0 a_1 ... a_n s_n$, given s_0, fulfills some goal condition γ in s_n. In a nondeterministic environment α_n is a *conformant plan*, if for each trajectory $s'_0 a_1 ... a_n s'_n$ (assuming $|R(s'_i, a_{i+1})| > 0$ for each s'_i) γ is fulfilled in s'_n, s.t. $s'_0 = s_0$, cf. (Son et al. 2023).

Norms. A *normative system* is a set of norms commonly represented in a deontic logic. Many deontic logics, like the widely used *Standard Deontic Logic (SDL)*, extend classical logic with deontic operators (Gabbay and Horty 2013), including obligation $\mathbf{O}(p)$ (it is obligatory that p), permission $\mathbf{P}(p)$ (it is permissible that p), and prohibition $\mathbf{F}(p)$ (it is forbidden that p). Typically, $\mathbf{O}$ is the primitive operator and

$$\mathbf{P}(p) := \neg\mathbf{O}(\neg p), \qquad \mathbf{F}(p) := \mathbf{O}(\neg p).$$

defines the others. As we are primarily concerned with temporal domains, we will interpret all norms as so-called *maintenance norms*. Akin to a *maintenance obligation* (Governatori et al. 2007), a maintenance norm Φ is fulfilled if the agent complies with it at every time point; Φ is violated if at some time point the agent does not comply with it.



(a) Deterministic (b) Nondeterministic

Figure 1: Sample instances of the Gardener game

In deontic logic, norms do not simply represent hard constraints but rather indicate ideal actions or states, allowing for norm violations or deviations. In ASP, this can be reflected by weak constraints, which allow to assign penalties to constraint violations. Paradoxes like contrary-to-duty obligations (Prakken and Sergot 1996), where the agent has to oblige to a norm only if another norm is violated, underline the complexity of deontic logic. ASP is well-equipped to handle such challenging scenarios, as evidenced by Hatschka, Ciabattoni, and Eiter (2023) and Governatori (2024).

Reinforcement Learning (RL). The basic idea of RL is to train an agent by trial-and-error interactions in an environment giving rewards or penalties depending on the action outcome (Barto 1997). During training, the agent aims to maximize its cumulative reward over time and occasionally picks entirely random actions with a certain probability to increase exploration. After training, the agent picks for any state the action with the expectedly maximum reward. The respective mapping $\pi : S \rightarrow A$ is called the agent's *policy*. We may assume that the agent stops executing actions once a goal condition γ is reached; in case of $\gamma = \bot$ it never stops.

A popular realization of RL is *Q-Learning* using a *Q-Table*, which holds for each state-action pair (s, a) a value (real number) that is updated during training depending on the rewards/penalties received. Eventually this leaves the agent with a policy that selects an action a in state s using a function $Q : S \times A \rightarrow \mathbb{R}$ such that $a = \text{argmax}_{a'} Q(s, a')$. Notably, Q-learning allows for a ranking of actions given a state, which is a requirement for our framework, as this enables encoding the RL preferences into an ASP program.

Example 1 As a running example, we consider a simple, 2-dimensional board game we call *Gardener*. In Gardener, every instance contains an agent whose goal is to reach a given target cell as quickly as possible, where the available actions are $A = \{north, east, south, west\}$, each moving the agent by one cell at a time in the according direction. Furthermore, there may be walls (visualized as black cells in Figure 1) that the agent cannot traverse, and cells can contain vegetation (plants) or wildlife (frogs). When the agent moves onto such a cell, it kills the plant resp. frog. Frogs move arbitrarily at each time step with an action in A, while plants don't. Hence, instances without frogs induce deterministic transition systems and instances with frogs nondeterministic transition systems; Figure 1 shows some samples.

A simple Q-learning realization to reach the target cell as soon as possible could yield a policy π_f by giving the agent

during training a high reward when reaching the target and punishing it for each move with a small amount.

Policy Fixing

Assume it is noticed that during the execution of its policy π, the agent violates the maintenance norm Φ at some time point t. Now the following problem arises:

Problem statement. Can we alter policy π, so that following the new policy π' the agent's subsequent actions minimize further violations of Φ while accounting for the action preferences of π, balancing norm violations and policy adherence?

The notion of a *policy fix* formally captures this as follows. Suppose that a violation of Φ in a state s at any time t' has a real-valued penalty $p(\Phi, s, t') \geq 0$, and that $\pi' \preceq_Q \pi$ is the preorder of policies according to the learned Q-table, i.e., for each state s, $\pi'(s)$ is ranked better or equal than $\pi(s)$.

Definition 1 *Given a transition system T, a maintenance norm Φ, an agent policy π, and a time point t, a* policy fix *for π at t is a policy π' such that*

$$\pi' = \text{argmin}^{\preceq_Q}_{\pi'' \in \Pi} \max_{\sigma \in Tr(\pi'',t)} \sum_{t'>t} p(\Phi, s_{t'}, t'), \quad (1)$$

where Π is the set of all possible policies and $Tr(\pi'', t)$ denotes the set of all trajectories σ whose suffix from t, i.e., $s_t, a_{t+1}, s_{t+1}, \ldots$, complies with π''.

Informally, for selecting π' we must consider the worst case total norm violation by π''; ideally, it is zero, thus most preferred. We call such a policy fix *strict*. Among policies with equal penalty, those adhering closer to the Q-table are preferred, which intuitively aids in keeping a good reward.

Instead of prioritizing norm obedience, we may adjust equation (1) to jointly optimize reward and norms.

Definition 2 *A* utility-based policy fix *is a policy fix with "$\text{argmin}^{\preceq_Q}_{\pi'' \in \Pi}$" replaced by "$\text{argmax}_{\pi'' \in \Pi} r(\pi'', s_t, t) -$ " in (1), where $r(\pi'', s_t, t)$ is the expected reward of π'' from t onwards, i.e., norm violation acts as reward discount.*

Solving (1) amounts to a special conformant planning problem, as we must find a course of actions that ensures the agent will always reach some goal state. However, conformant planning is expensive (EXPSPACE-complete (Rintanen 2004)), and it is already Σ_2^P-complete when restricted to plans of polynomial length in conventional settings, cf. (Baral, Kreinovich, and Trejo 2000). Considering real-time demands, we thus resort to approximate policy fixes of bounded length.

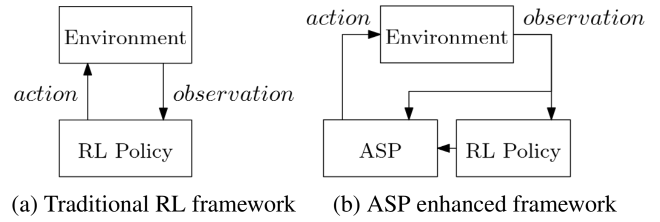
Definition 3 *A k-*policy fix *for a policy π at time t is a policy fix π' for π at t with goal condition γ relaxed to allow stopping after k steps.*

Thus, a k-policy fix is a sequence $a_1, \ldots, a_n$ of $n \leq k$ actions optimal for the next n steps. Hence, we resort to repeated computation of conformant plans of limited length. We may compute a new plan after k steps, or after $i < k$ steps (in the extremal case, at each step), balancing computational cost and increased norm compliance assurance.

Example 2 (cont'd) Suppose that following π_f learned from the instance in Fig. 1a, the agent first moves up and then right until it reaches the target; it then kills a plant on its way. Similarly, for the instance in Fig. 1b, the agent does not consider the whereabouts of the frog when inching closer to the target and will very likely kill the frog.

Suppose the maintenance obligation $\mathbf{O}(do_not_kill)$ is added, obliging the agent to avoid killing living entities. Policy π_f clearly violates this norm. For Fig. 1a, a strict 2-policy fix is feasible. For Fig. 1b a strict 2-policy prevents norm violations but may miss the target. In this case, an optimization of rewards and norm obedience might be preferred.

While in Examples 1 and 2 we can retrain the policy in a reasonable time. However, it serves to illustrate how to build a framework utilizing ASP to alter π_f by computing a k-policy fix at each time step by default.



(a) Traditional RL framework (b) ASP enhanced framework

Figure 2: Decision-making framework

ASP Framework

We propose a framework that creates a policy fix for RL policies by outsourcing norm compliance to a logic component. Figure 2a shows a traditional RL decision-making framework. Here, environment observations are sent to the RL policy, which then selects an action for the agent to execute.

In contrast, the decision-making process of the proposed framework, shown in Figure 2b, feeds the observations in addition to an ASP component. Together with the preference information of the RL policy, the latter selects an action for execution based on a conformant plan of length k. At each time step, the framework creates a k-policy fix and executes the next action according to the altered policy. For simplicity, we tacitly assume that a norm violation triggering the ASP component already occurred in all scenarios that we consider.

Answer Set Programming (ASP) is a declarative approach to problem solving with many applications (Brewka, Eiter, and Truszczyński 2011; Lifschitz 2019). It revolves around encoding problems into logic programs, i.e., sets of rules

$$a_1 \mid \ldots \mid a_l \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n \quad l, m, n \geq 0,$$

where all a_i's, b_j's and c_k's are atoms and *not* is default negation. If $l = 0$, the rule is a constraint; if $l > 1$, any a_i may be derived; such disjunctive rules allow for nondeterminism.

The semantics of an ASP program is given by special classical models called *answer sets*, which are determined by the stable model semantics (Gelfond and Lifschitz 1991). Using an efficient ASP solver like *clingo* (Gebser et al. 2014) or *DLV2* (Alviano et al. 2017) some answer set(s) can be computed and solutions to an encoded problem extracted from them. Notably, ASP offers many further constructs like choice rules, aggregates, and for optimization *weak constraints* (Calimeri et al. 2020). The latter are rules of the form

$: \sim B.\,[W]$. that can be violated (i.e., B is satisfied) at the cost of a (possibly prioritized) weight W. An ASP solver then selects answer sets with optimal cost.

These features and the ease of expressing inertia in action execution make ASP a powerful and well-suited tool for solving expressive planning problems.

Program Composition. The ASP component uses a program P to create conformant plans for the agent that consider additional norms and the action preferences of the RL policy. Given the current state, the preferences can be encoded as an ordinal ranking of actions in P. In case of multiple answer sets, an answer set solver can then optimize for the policy preferences. We propose to structure the program P following the "guess and check" methodology (Eiter, Ianni, and Krennwallner 2009). Specifically, P consists of two subprograms, P_{gen} and P_{check}, which intuitively work as follows:

1. P_{gen} uses choice rules (i.e., disjunction in rule heads) to create possible trajectories for the agent (defined by the deterministic consequences of its actions);
2. P_{check} checks whether a candidate trajectory qualifies as a solution and eliminates trajectories in which a norm violation is possible (by checking all possible guesses of the nondeterministic environment).

Atoms. For describing P_{gen} and P_{check}, we will use abstractions of atoms as follows:

- State atoms $s(T, X_1, ...X_i)$, with $i \geq 0$; state specific properties of the agent or the environment at a time T.
- Action atoms $a(T)$; an action the agent or the environment takes at a time T.
- Static atoms $f(X_1, ..., X_i)$, $i \geq 0$; static properties of the agent or the environment.
- Helper atoms $o(X_1, ..., X_i)$, $i \geq 0$; used in advanced encoding techniques such as the *saturation technique* below.

For better readability, we omit arguments of these atoms that are irrelevant to the given context. For example, we simply write $s(0)$ as a stand-in for a state atom at time 0.

Rules. We refer to some types of rules that follow a specific pattern in how they are composed:

- Facts: f; describe static properties of the agent, the environment, or a specific instance.
- Initial state facts: $s(0)$; describe the initial state of the agent or the environment.
- Action rules: $a_1(T) \mid a_2(T) \mid ... \mid a_i(T) \leftarrow T \leq k$, with $i \geq 1$; describe the possible actions the agent or the environment can execute at each time T.
- Transition rules: $s_1(T) \leftarrow a(T), s_2(T-1), ..., s_i(T-1)$, with $i \geq 2$; describe a change that occurs in the domain, when progressing from time $T-1$ to T.
- Constraint rules: $\leftarrow f_1, ..., f_i, s_1(T), ..., s_j(T)$ with $i, j \geq 0$ and $i + j \geq 1$; describe a combination of fact and state atoms not allowed to be true at time T.
- Norm fulfillment rules: $o(T) \leftarrow f_1, ..., f_i, s_1(T), ..., s_j(T), o(T-1)$, with $i, j \geq 0$ and $i + j \geq 1$; describe a combination of facts and state atoms that must be true at each time T for a norm to be complied with.

Listing 1: Excerpt of P_{gen}

```
1 plant(4,4).
2 player(0,2,6).
3 p_n(T)|p_s(T)|p_w(T)|p_e(T) :- T=1..k.
4 player(T,C',R) :- player(T-1,C,R),
⋮     p_w(T), C'=C-1.
8 :~ player(T,C,R), plant(C,R). [1@2]
9 #maximize {R@1,T : reward(R,A,T)}.
```

Subprogram $\mathbf{P_{gen}}$. The goal of P_{gen} is to create an answer set M_{gen} that represents a guess for the action sequence executed by the agent. In general, this is achieved with the following facts and rule composition:

- Static facts that describe the static properties of the agent, the environment, and the specific instance. In addition,
- Initial state facts for state atoms that do not rely on nondeterministic behavior.
- Most importantly, a disjunctive action rule that guesses a distinct action for each time $T \leq k$.
- Transition rules dependent on the guessed actions and state atoms not reliant on nondeterministic behavior.
- Finally, constraints and weak constraints to exclude answer sets resp. make them less attractive. Crucially, in P_{gen}, these constraints can only contain state atoms that do not depend on nondeterministic behavior.

Example 3 (cont'd) An excerpt of a concrete implementation of P_{gen} for the Gardener domain is shown in Listing 1. The plant's position is encoded in Line 1 by a fact and the agent's starting position by an initial state fact in Line 2. Line 3 contains the action rule, while Lines 4-7 contain the transition rules encoding the effects of the chosen action. Line 8 contains a weak constraint that assigns penalty 1 at priority level 2 whenever the agent is on a cell with a plant. Finally, Line 9 contains an optimization statement that prompts the solver to adhere to the RL policy's preferences at lower priority (level 1), which are encoded using the $reward$ atom.

Subprogram $\mathbf{P_{check}}$. The purpose of P_{check} is to check whether M_{gen} is a conformant plan regarding any additional norms that are dependent on the nondeterministic behavior of the environment. To guess this behavior, the facts and rules of the basis of P_{check} are similar to P_{gen}:

- Initial state facts of the state atoms that are influenced by the actions of the environment.
- A disjunctive action rule for each nondeterministic element in the environment.
- Transition rules for the effects of the guessed actions.
- Rules for checking norm fulfilment and for penalization.

Intuitively, one might be tempted to use simple constraints to enforce that norms are fulfilled. However, doing so would answer the question "Is it possible for the agent to not violate the norm, i.e., fulfill it given its next k actions?"; in contrast, we are interested in the the question "Is there *no possibility* for the agent *to violate* the norm given its next k actions?". To answer the latter, P_{check} employs the so-called saturation technique (Eiter, Ianni, and Krennwallner 2009).

Listing 2: Excerpt of P_{check}

```
10  frog(0,6,2).
11  f_n(T)|f_s(T)|f_w(T)|f_e(T) :- T=1..k.
12  frog(T,C',R) :- frog(T-1,C,R), f_w(T),
 ⋮       C'=C-1.
16  ok(0).
17  ok(T) :- player(T,C,R), frog(T,C',R'),
 ⋮       C'!=C, ok(T-1).
19  :- not sat.
20  sat :- ok(k).
21  f_n(T) :- T = 1..k, sat.
 ⋮
25  sat :- frog(T,C,R), wall(C,R).
26  good(T) | bad(T) :- T=1..k.
27  ok(T) :- bad(T).
28  #minimize {1@3,T : bad(T)}.
```

Saturation technique. The saturation technique allows for solving problems relying on the universal quantifier, as it enables encoding of for all conditions. To that end, we design a program P_{sat} and an atom set M_{sat} such that P_{sat} has the single answer set M_{sat} if the property $\mathcal{P}$ to check (which has coNP complexity; in our case, plan conformance) holds and other answer sets otherwise. Each candidate answer set M on which $\mathcal{P}$ holds can be "saturated" to M_{sat} by including other atoms, using a special atom sat; if $\mathcal{P}$ does not hold, saturation fails and M is an answer set without sat.

To use this technique for checking the norm-compliance of all possible evolutions in the environment, we introduce a norm-fulfillment rule for each obligation. These rules derive a helper atom $ok(T)$ only if the obligations are fulfilled in the current and recursively all previous time steps. Hence, if $ok(k)$ is in the candidate answer set M, the norm compliance holds for plan length k, and we saturate to M_{sat}. By adding $\leftarrow not\ sat$, we ensure that a candidate answer set of P is acceptable iff it contains M_{sat}, thus rejecting all action sequences which admit a norm violation. Notably, empty heads are in P_{check} disallowed and replaced with sat. Impossible states etc. are then safely eliminated by saturation to M_{sat}.

Example 4 (cont'd) Listing 2 shows an excerpt of P_{check} for checking norm compliance in the nondeterministic version of Gardener. Here, Line 10 introduces an initial state fact, while we create a guess for the nondeterminism in Line 11 with an action rule. The state transition rules in Lines 12-15 capture the effects of this rule. Line 16 initializes the helper atom for the checked norm, while the norm fulfillment rules in Lines 17-18 check the compliance in each time step. Line 19 is the saturation constraint, and the rules in Lines 20-25 are the saturation part. Specifically, Line 25 is a saturation rule for impossible moves of the frog.

Applying the saturation technique, we cannot use weak constraints naively to penalize norm violations, as the saturated answer set M_{sat} may falsify the penalty count. The penalization can be achieved by a template shown in Lines 26-28 of Listing 2: a guess for $bad(T)$ in Line 26 offers an alternative to derive $ok(T)$ in Line 27; Line 28 then minimizes the number of occurrences of bad-atoms over the plan length.

Program Optimization

The quality of the conformant plans generated with the logic program from above depends on the chosen length k. However, increasing k also comes at a computational cost. Hence, making the encoding and program evaluation highly efficient is important. To this end, we present optimization techniques that aid in boosting computational performance.

Multi-shot solving. *Clingo*'s multi-shot solving (MSS) capabilities refer to features that enable the answer set solver to deal with continuously changing programs using the `#external` directive (Gebser et al. 2019). Atoms defined with it must specify all potential values for their arguments and are grounded, i.e., instantiated, along with the other rules of the program. During grounding–a known bottleneck of ASP performance–these atoms are viewed as inputs, whose truth value can be set via *clingo*'s API. Intuitively, MSS allows for dynamically (de-)activating rules or facts between solving steps, enabling an incremental problem-solving approach.

Another use case important for our framework is grounding all possible states of a domain for the initial computation step using input atoms defined as `#external`. These atoms must describe all potential arguments already before grounding. Hence, each possible state can be described purely by adapting the input atoms via *clingo*'s API. This technique requires grounding only in the initial computation step, at the cost of increased computation time. Thus keeping the number of ground rules generated during grounding as low as possible is desirable. We next discuss a technique that can reduce the number of rules by lowering predicate arities.

Dimensionality Reduction. To describe connected properties such as attributes of an object, it is natural to use a single atom with multiple arguments, each describing a different property; e.g., $player(T, C, R)$ to describe the agent's position at some time. Reducing the dimensionality of such atoms by splitting them into multiple atoms, thus describing the properties individually, can greatly benefit performance. Importantly, the properties in question must be independent; the atom above may be split into $p_col(T, C)$ and $p_row(T, R)$.

Reducing an atom's dimensionality means to reduce its arity by at least one. Combined with MSS, this aids reducing the number of grounded rules that involve comparisons between atoms concerning only some arguments of the atoms.

In grid-style environments like in Gardener, the concerned comparisons are often position-related and include checking whether two entities are on the same/a different cell; whether some entity is above/below or to the left/right of another entity; or even computing the distance between two entities.

Example 5 (cont'd) Suppose we want to derive the helper atom $ok(T)$ if the player and the frog are not on the same cell at a time T. Traditionally, this comparison is encoded as in Line 17 (for column comparison) of Listing 2. Assuming that $player(T, C, R)$ and $frog(T, C', R')$ are external input atoms defined over the whole grid, we would ground $(c{\cdot}r)^2$ rules for each time step, where c is the number of columns and r is the number of rows. However, the dimensionality of atoms $player$ and $frog$ can be reduced, which results in a linearized version (in terms of grid dimensions) of the rule:

$$ok(T) \leftarrow p_col(T, C), f_col(T, C'), C \neq C'.$$

Assuming that $p_col(T, C)$ and $f_col(T, C')$ are external input atoms defined over the whole grid, this encoding results in c^2 rules generated in grounding for each time step; which is a significant reduction of the number of rules.

Windowing. A further optimization is what we refer to as *windowing*. Informally, windowing is encoding only a part of the global state that is (most) relevant for the agent at a point in time in the logic program. From a technical perspective, this is also realized with the `#external` directive. Here, the input atoms are chosen such that, by changing them via *clingo*'s API, one could potentially describe every possible window. For grid-style applications like Gardener, the window may be chosen as the state of the cells in radius r around the agent's current position; in Gardener, for the next $r/2$ steps the global state beyond the window does not matter.

The advantages of windowing for planning are the constant maximum number of rules to ground and that expensive inferences from outside the window can be ignored. This is apparent with growing instance size: after setup, windowing enables nearly constant computation time. It works best if the agent's decisions and norm violations depend on a window around the agent, e.g., the neighbourhood cells, as planning loses perspective on everything outside the window.

Experiments

In this section, we showcase the practical applicability of the framework in multiple domains, including deterministic and nondeterministic environments. For evaluation, we compare the norm compliance of using a RL policy π (which is unaware of additional norms) against the use of π in combination with our framework. In particular, we are interested in the effect of the horizon k on norm violations. We run each test instance multiple times with different parameters for the window size considered by the agent (given as a radius r) and the horizon k. We expect to confirm that:

- In general, a bigger horizon k and window radius r lead to less norm violations while the computation time increases.
- Increasing k or r disproportionally from each other will result in worse norm compliance than a balanced approach.
- Any window radius r has a practical sealing for the growth of step computation time, regardless of the instance size.
- The startup and average computation time increase with k and r respectively. However, a sensible balance of the parameters should still work in real-time scenarios.

Experimental Setup. We use utility-based policy fixing to prevent that the agent gets stuck by unsolvability (i.e., finds no strict k-policy fix) and ease achieving the goal. We use the ASP solver clingo (version 5.7.1) to solve the ASP program of the framework. Further code and the RL implementations are in Python. The experiments were run on a Linux server with two Intel Xeon CPU E5-2650 v4 (12 cores @ 2.20GHz, no hyperthreading) and 256GB RAM. Our programs and all experimental data are available in our Git repository.

Experiment 1: Gardener. We randomly generated 80 instances of Gardener, for both the deterministic and the nondeterministic case. The 80 instances are evenly split into 20 quadratic grid sizes of length $10, 25, 50$ and 100. For every grid size, we populated the cells of 10 instances having 25% walls with 5% respectively 10% living entities, half of which are frogs in nondeterministic environments.

Config	1-Step kills plants / frogs	1-Step avg. / startup time in ms	2-Step kills plants / frogs	2-Step avg. / startup time in ms
2500 walls, 1000 plants and 0 frogs				
RL	11.2 / 0	0 / 0	11.2 / 0	0 / 0
r=3, k=1	6.0 / 0	179 / 180	NA	NA
r=3, k=2	5.4 / 0	182 / 196	4.8 / 0	91 / 187
r=3, k=4	3.5 / 0	198 / 256	3.3 / 0	98 / 248
r=3, k=6	3.5 / 0	283 / 414	3.3 / 0	140 / 413
r=5, k=1	6.0 / 0	268 / 258	NA	NA
r=5, k=2	5.3 / 0	270 / 266	4.8 / 0	134 / 264
r=5, k=4	3.5 / 0	283 / 333	3.6 / 0	143 / 336
r=5, k=6	2.7 / 0	411 / 572	3.1 / 0	203 / 569
2500 walls, 250 plants and 250 frogs				
RL	4.00 / 2.80	47 / 46	4.00 / 2.80	47 / 46
r=3, k=1	2.02 / 1.24	284 / 359	NA	NA
r=3, k=2	1.74 / 1.14	297 / 505	1.56 / 1.24	173 / 507
r=3, k=4	0.96 / 0.78	469 / 1360	0.92 / 0.80	257 / 1367
r=3, k=6	0.86 / 0.64	1438 / 3545	0.64 / 0.82	727 / 3826
r=5, k=1	1.90 / 1.12	424 / 954	NA	NA
r=5, k=2	1.80 / 0.82	508 / 2312	1.70 / 1.08	280 / 2381
r=5, k=4	0.90 / 0.76	1424 / 12592	0.84 / 0.74	720 / 12983
r=5, k=6	1.02 / 0.66	6528 / 54842	0.98 / 0.74	3334 / 55767

Table 1: Gardener *big/small-d-100-0x*, x =*01..10* (100×100)

In the policy fix, the penalty for killing a living entity is higher than any action reward. Plans admitting repeated states are penalized to discourage looping in norm-compliant trajectories. An extra reward for reaching the target is given, and rewards for obeying policy preference inrease on cell revisits. Earlier violations and norm adherence are weighted higher to discourage immediate violations or policy deviations.

Table 1 shows the results for deterministic and nondeterministic instances on a 100×100 grid with varying r and k. The results indicate that the framework, even with $k = 1$ drastically increased norm compliance. Better results were achieved with varying r and k. Notably, when executing the first two steps of a k-policy the run time is (as expected) half, while norm compliance is not significantly worse.

Experiment 2: Pacman. We consider an RL agent for the popular video game *Pacman*, a grid game where the agent (Pacman) navigates a maze and aims to collect as many points as possible. Pacman earns 10 points for eating a "food pellet" (i.e., a small dot in a cell) and loses 1 point for each move. Pacman wins if he ate all food pellets, which earns him 500 points; he loses when hitting a non-scared ghost. Ghosts are temporarily scared after Pacman ate a "power pellet" (i.e., a big dot). Eating a scared ghost earns Pacman 200 points. The number of ghosts varies depending on the instance.

Like Neufeld et al. (2022) and Noothigattu et al. (2018), we use two maintenance obligations: *vegan*, obliging Pacman to not eat ghosts, and *vegetarian*, obliging Pacman only eat the blue ghost. We extended the Berkeley AI code for Pacman (DeNero, Klein, and Abbeel 2014) and use associated instances: *smallClassic* (grid size 20×7; 2 ghosts), *mediumClassic* (20×11;2) and *originalClassic* (28×27;4). Instances were tested 1000 times with two RL policies: *hungry*, where Pacman can see ghosts are scared, and *simple*, where he can't.

The policy fix weighs norm violations higher than rewards.

Config	Vegan ghosts eaten blue / other	Vegan % wins [avg. score]	Vegetarian ghosts eaten blue / other	Vegetarian % wins [avg. score]
mediumClassic				
RL	0.83 / 0.82	92 [1559]	0.83 / 0.82	92 [1559]
r=3, k=1	0.02 / 0.03	91 [1204]	0.78 / 0.00	90 [1343]
r=3, k=2	0.01 / 0.01	92 [1224]	0.82 / 0.00	90 [1345]
r=3, k=4	0.02 / 0.01	94 [1244]	0.80 / 0.00	89 [1344]
r=3, k=6	0.01 / 0.01	94 [1242]	0.78 / 0.00	89 [1328]
r=5, k=1	0.01 / 0.01	91 [1209]	0.79 / 0.00	91 [1362]
r=5, k=2	0.02 / 0.02	93 [1236]	0.84 / 0.00	90 [1364]
r=5, k=4	0.01 / 0.01	95 [1262]	0.82 / 0.00	90 [1352]
r=5, k=6	0.01 / 0.01	95 [1257]	0.77 / 0.00	90 [1340]
originalClassic				
RL	0.60 / 1.68	81 [2581]	0.60 / 1.68	81 [2581]
r=3, k=1	0.02 / 0.05	76 [2014]	0.56 / 0.03	78 [2139]
r=3, k=2	0.01 / 0.04	79 [2056]	0.53 / 0.03	80 [2181]
r=3, k=4	0.01 / 0.04	83 [2124]	0.53 / 0.03	81 [2180]
r=3, k=6	0.01 / 0.04	83 [2120]	0.52 / 0.04	81 [2187]
r=5, k=1	0.02 / 0.07	78 [2023]	0.55 / 0.02	78 [2145]
r=5, k=2	0.02 / 0.06	83 [2126]	0.52 / 0.04	82 [2202]
r=5, k=4	0.01 / 0.04	85 [2156]	0.52 / 0.01	82 [2210]
r=5, k=6	0.01 / 0.02	88 [2212]	0.54 / 0.02	83 [2243]

Table 2: Pacman (hungry version)

The action reward is based on policy preference; earlier violations are weighted higher to discourage immediate violations.

Table 2 shows the results for *mediumClassic* and *originalClassic* under *hungry*, with varying parameters r and k. The pure RL approach, which is norm-agnostic, serves as a baseline. In *originalClassic*, Pacman eats under *vegan* 2.28 ghosts on average. Our framework raised norm compliance here drastically, with Pacman eating at most 0.09 ghosts on average. Increasing r resp. k in general payed off compared to 1-step look ahead, achieving the best results for $r = 5, k = 6$. In *mediumClassic*, the picture was similar; even achieving full norm compliance under *vegetarian*.

Throughout, the average score was expectedly lower than in the RL baseline, but wins remained steady or rose with increased k. The average computation time for a utility-based k-policy fix stayed within $54ms$. However, for small k it was much faster, though increasing progressively.

As the *simple* RL policy is almost norm-compliant by itself, our framework improved norm compliance only marginally.

Experiment 3: CCN Caching. To showcase our framework beyond grid games, we apply it for caching on a router in a Content Centric Network (CCN). CCNs focus on delivering data by its content or name, rather than by its location as in IP-based networks (Shinde and Chaware 2018). Deciding which packages a router in a CCN should store is a non-trivial problem. The success of the applied caching strategy (measured via *cache hits*) can significantly impact network performance. *Meta policies*, which dynamically switch between pre-defined standard polices like LRU, FIFO, or Random, were successfully generated using logic (Beck et al. 2017) or RL based approaches (Iqbal and Asaduzzaman 2023).

We consider a network of 5 consumers, 5 intermediate routers, and 3 content providers, where content has type *movie*, *news* or *gov*. Using the sota CCN simulator *ndnSIM 2.9* (Mastorakis, Afanasyev, and Zhang 2017), we simulate this network for 30 secs on 5 instances differing in type and frequency of consumer requests while observing the cache

instance [packages]	LRU norm violations low / high	LRU cache hits	ASP + LRU norm violations low / high	ASP + LRU cache hits
scenario 1 [385]	504 / 20	2	52 / 0	2
scenario 2 [358]	1419 / 20	2	76 / 0	2
scenario 3 [289]	271 / 95	7	52 / 88	8
scenario 4 [389]	744 / 53	4	62 / 0	5
scenario 5 [419]	2352 / 99	10	128 / 0	7

Table 3: CCN Caching

(size 18) of a router that is a bottleneck to all traffic. We define two norms, obliging the router (i) to not cache low priority content (*news*) beyond 10 time steps and (ii) to not remove high priority content (*gov*) less than 20 time steps old. We assume LRU is the current policy, selected by a meta-policy, and compute utility-based k-policy fixes for k=3.

In the policy fix, action reward is based on policy preference and norm violations are penalized by content age.

The results in Table 3 show a drastic reduction of norm violations for low and high priority packages by our framework. In scenario 3, the 88 in case high is due to many priority package requests, which fill up the cache, leading necessarily to norm violations. Interestingly, the cache performance decreased by policy fixing only in one scenario. An extensive study of the behaviour remains for future work.

Conclusion

As easily seen, Neufeld et al.'s (2022) normative supervisor amounts to our framework for $k = 1$ and $r > 2$. For Gardener, norm compliance improved largely with k. For Pacman, $k = 1$ already yields high norm compliance, but wins tend to increase with k. Compared to the RL baselines, the variance of norm violations by our framework is much smaller.

Deterministic domains expectedly proved easier to compute, as they do not rely on a coNP check. Computation times further suggest greater scalability compared to nondeterministic domains. However, mindfully choosing the horizon k enables the application in nondeterministic domains even in real-time scenarios, as shown in the Pacman example.

As *telingo* (Cabalar et al. 2019) is a dedicated solver for temporal ASP, using it for policy fixing instead of clingo seems suggestive. However, *telingo* does not support backpropagation in time (i.e., deriving truth values of atoms in the past), which is crucial for our encodings in nondeterministic domains. Moreover, as *telingo* lacks a Python API and support for MSS, our optimizations can not be fully exploited. Tests on deterministic instances showed no performance gain.

Outlook. For dealing with coNP subproblems as in conformant planning, other approaches than the saturation technique may be used. *ASP(Q)* (Amendola et al. 2022) allows calls to a QBF solver in rule bodies, which enables conformancy checking of plans. The recent *ASP recipe* system (Alviano, Cirimele, and Reiners 2023) supports glueing ASP programs together via operations. An ASP program may compute and enumerate all potential plans, while a second one could receive the plans as an input and address the norm compliance requirement. It would be interesting to see whether our framework could benefit from using one of these systems.

Acknowledgements

This research was funded in whole or in part by the Vienna Science and Technology Fund (WWTF) project ICT22-023 and the Austrian Science Fund (FWF) 10.55776/COE12.

References

Alviano, M.; Calimeri, F.; Dodaro, C.; Fuscà, D.; Leone, N.; Perri, S.; Ricca, F.; Veltri, P.; and Zangari, J. 2017. The ASP System DLV2. In *LPNMR*, volume 10377 of *Lecture Notes in Computer Science*, 215–221. Springer.

Alviano, M.; Cirimele, D.; and Reiners, L. A. R. 2023. Introducing ASP recipes and ASP Chef. In *ICLP Workshops*, volume 3437 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Amendola, G.; Cuteri, B.; Ricca, F.; and Truszczynski, M. 2022. Solving Problems in the Polynomial Hierarchy with ASP(Q). In *LPNMR*, volume 13416 of *Lecture Notes in Computer Science*, 373–386. Springer.

Aminof, B.; Giacomo, G. D.; Murano, A.; and Rubin, S. 2019. Planning under LTL Environment Specifications. *Proceedings of the International Conference on Automated Planning and Scheduling*, 29: 31–39.

Baral, C.; Kreinovich, V.; and Trejo, R. 2000. Computational Complexity of Planning and Approximate Planning in the Presence of Incompleteness. *Artificial Intelligence*, 122(1-2): 241–267.

Barto, A. G. 1997. Reinforcement Learning. In *Neural Systems for Control*, 7–30. Elsevier. ISBN 978-0-12-526430-3.

Beck, H.; Bierbaumer, B.; Dao-Tran, M.; Eiter, T.; Hellwagner, H.; and Schekotihin, K. 2017. Stream reasoning-based control of caching strategies in CCN routers. In *ICC*, 1–6. IEEE.

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer Set Programming at a Glance. *Communications of the ACM*, 54(12): 92–103.

Cabalar, P.; Kaminski, R.; Morkisch, P.; and Schaub, T. 2019. telingo = ASP + Time. In *LPNMR*, volume 11481 of *Lecture Notes in Computer Science*, 256–269. Springer.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; and Schaub, T. 2020. ASP-Core-2 Input Language Format. *Theory Pract. Log. Program.*, 20(2): 294–309.

DeNero, J.; Klein, D.; and Abbeel, P. 2014. UC Berkeley CS188 Intro to AICourse Materials. http://ai.berkeley.edu/project_overview.html. Accessed: 2024-08-12.

Eiter, T.; Ianni, G.; and Krennwallner, T. 2009. Answer Set Programming: A Primer. In *Reasoning Web*, volume 5689 of *Lecture Notes in Computer Science*, 40–110. Springer.

Gabbay, D. M.; and Horty, J., eds. 2013. *Handbook of Deontic Logic and Normative Systems*. London: College Publications. ISBN 978-1-84890-132-2.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. Clingo = ASP + Control: Preliminary Report. arXiv:1405.3694.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-Shot ASP Solving with Clingo. *Theory and Practice of Logic Programming*, 19(1): 27–82.

Gelfond, M.; and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3-4): 365–385.

Giordano, L.; Martelli, A.; and Dupré, D. T. 2013. Temporal deontic action logic for the verification of compliance to norms in ASP. In *ICAIL*, 53–62. ACM.

Governatori, G. 2024. An ASP Implementation of Defeasible Deontic Logic. *KI - Künstliche Intelligenz*.

Governatori, G.; Hulstijn, J.; Riveret, R.; and Rotolo, A. 2007. Characterising Deadlines in Temporal Modal Defeasible Logic. In *Australian Conference on Artificial Intelligence*, volume 4830 of *Lecture Notes in Computer Science*, 486–496. Springer.

Hatschka, C.; Ciabattoni, A.; and Eiter, T. 2023. Deontic Paradoxes in ASP with Weak Constraints. *Electronic Proceedings in Theoretical Computer Science*, 385: 367–380.

Iqbal, S. M. A.; and Asaduzzaman. 2023. Cache-MAB: A Reinforcement Learning-Based Hybrid Caching Scheme in Named Data Networks. *Future Generation Computer Systems*, 147: 163–178.

Kasenberg, D.; and Scheutz, M. 2018. Norm Conflict Resolution in Stochastic Domains. In *AAAI*, 85–92. AAAI Press.

Kiran, B. R.; Sobh, I.; Talpaert, V.; Mannion, P.; Sallab, A. A. A.; Yogamani, S.; and Perez, P. 2022. Deep Reinforcement Learning for Autonomous Driving: A Survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(6): 4909–4926.

Kober, J.; Bagnell, J. A.; and Peters, J. 2013. Reinforcement Learning in Robotics: A Survey. *The International Journal of Robotics Research*, 32(11): 1238–1274.

Lifschitz, V. 2019. *Answer Set Programming*. Cham: Springer International Publishing. ISBN 978-3-030-24657-0 978-3-030-24658-7.

Mastorakis, S.; Afanasyev, A.; and Zhang, L. 2017. On the Evolution of ndnSIM: An Open-Source Simulator for NDN Experimentation. *ACM SIGCOMM Computer Communication Review*, 47(3): 19–33.

Neufeld, E. A.; Bartocci, E.; Ciabattoni, A.; and Governatori, G. 2022. Enforcing Ethical Goals over Reinforcement-Learning Policies. *Ethics and Information Technology*, 24(4): 43.

Noothigattu, R.; Bouneffouf, D.; Mattei, N.; Chandra, R.; Madan, P.; Varshney, K.; Campbell, M.; Singh, M.; and Rossi, F. 2018. Interpretable Multi-Objective Reinforcement Learning through Policy Orchestration. arXiv:1809.08343.

Prakken, H.; and Sergot, M. 1996. Contrary-to-Duty Obligations. *Studia Logica*, 57(1): 91–115.

Rintanen, J. 2004. Complexity of Planning with Partial Observability. In *ICAPS*, 345–354. AAAI.

Shinde, A.; and Chaware, S. M. 2018. Content Centric Networks (CCN): A Survey. In *2018 2nd International Conference on 2018 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)*, 595–598.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play. *Science*, 362(6419): 1140–1144.

Son, T. C.; Pontelli, E.; Balduccini, M.; and Schaub, T. 2023. Answer Set Planning: A Survey. *Theory Pract. Log. Program.*, 23(1): 226–298.

Thananjeyan, B.; Balakrishna, A.; Nair, S.; Luo, M.; Srinivasan, K.; Hwang, M.; Gonzalez, J. E.; Ibarz, J.; Finn, C.; and Goldberg, K. 2021. Recovery RL: Safe Reinforcement Learning With Learned Recovery Zones. *IEEE Robotics and Automation Letters*, 6(3): 4915–4922.

Yu, C.; Liu, J.; Nemati, S.; and Yin, G. 2023. Reinforcement Learning in Healthcare: A Survey. *ACM Computing Surveys*, 55(1): 1–36.