The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

Safe Reinforcement Learning via Shielding

Mohammed Alshiekh,[1] **Roderick Bloem,**[2] **Rüdiger Ehlers,**[3]
Bettina Könighofer,[2] **Scott Niekum,**[1] **Ufuk Topcu** [1]
[1]University of Texas at Austin, 210 East 24th Street, Austin, Texas 78712, USA
[2]Graz University of Technology, Rechbauerstraße 12, 8010 Graz, Austria
[3]University of Bremen and DFKI GmbH, Bibliothekstraße 1, 28359 Bremen, Deutschland
{malshiekh, sniekum, utopcu}@utexas.edu, {roderick.bloem, bettina.koenighofer}@iaik.tugraz.at, rehlers@uni-bremen.de

Abstract

Reinforcement learning algorithms discover policies that maximize reward, but do not necessarily guarantee safety during learning or execution phases. We introduce a new approach to learn optimal policies while enforcing properties expressed in temporal logic. To this end, given the temporal logic specification that is to be obeyed by the learning system, we propose to synthesize a reactive system called a *shield*. The shield monitors the actions from the learner and corrects them only if the chosen action causes a violation of the specification. We discuss which requirements a shield must meet to preserve the convergence guarantees of the learner. Finally, we demonstrate the versatility of our approach on several challenging reinforcement learning scenarios.

Introduction

Advances in learning enabled a new paradigm for developing controllers for autonomous systems that accomplish complicated tasks in uncertain and dynamic environments. For example, in reinforcement learning (RL), an agent acts to optimize a long-term return that models the desired behavior for the agent and is revealed to it incrementally in a reward signal as it interacts with its environment (Sutton and Barto 1998). Increasing use of learning-based controllers in physical systems in the proximity of humans strengthens the concern of whether these systems will operate safely.

While convergence, optimality and data-efficiency of learning algorithms are relatively well understood, safety or more generally correctness during learning and execution of controllers has attracted significantly less attention. A number of different notions of safety were recently explored (García and Fernández 2015; Pecka and Svoboda 2014). We approach the problem of ensuring safety in reinforcement learning from a formal methods perspective. We begin with an unambiguous and rich set of specifications of what safety and more generally correctness mean. To this end, we adopt temporal logic as a specification language (Emerson 1990). For algorithmic purposes, we focus on the *safety* fragment of (linear) temporal logic (Manna and Pnueli 1995). We then investigate the question "how can we let a learning agent do whatever it is doing, and also monitor and interfere with its operation whenever absolutely needed in order to ensure safety?"

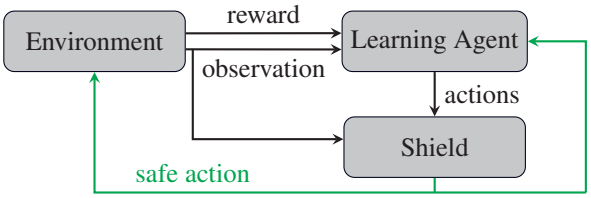

Figure 1: Shielded reinforcement learning

In this paper, we introduce *shielded learning*, a framework that allows applying machine learning to control systems in a way that the *correctness* of the system's execution against a given specification is assured during the learning and controller execution phases, regardless of how fast the learning process converges. The shield monitors the actions selected by the learning agent and corrects them if and only if the chosen action is unsafe.

In the traditional reinforcement learning setting, in every step, the learning agent chooses an action and sends it to the environment. The environment evolves according to the action and sends the agent an observation of its state and a reward associated with the underlying transition. The objective of the agent is to optimize the reward accumulated over this evolution.

Our approach introduces a *shield* into the traditional reinforcement learning setting. The shield is computed upfront from the safety part of the given system specification and an abstraction of the agent's environment dynamics. It ensures *safety* and *minimum interference*. With minimum interference we mean that the shield restricts the agent as little as possible and forbids actions only if they could endanger safe system behavior.

Shielding offers several pragmatic advantages: Even though the inner working of learning algorithms is often complex, shielding with respect to critical safety specifications may be manageable (as we demonstrate in upcoming sections). The algorithms we present for the computation of shields make relatively mild assumptions on the input-output structure of the learning algorithm (rather than its inner working). Consequently, the correctness guarantees

are agnostic—to an extent to be described precisely—to the learning algorithm of choice. Our setup introduces a clear boundary between the learning agent and the shield. This boundary helps to separate the concerns, e.g., safety and correctness on one side and convergence and optimality on the other and provides a basis for the convergence analysis of a shielded RL algorithm. Last but not least, the shielding framework is compatible with mechanisms such as function approximation, employed by learning algorithms in order to improve their scalability.

Related Work

Safety in RL. An exploration process is called *safe* if no undesirable states are ever visited, which can only be achieved through the incorporation of external knowledge (García and Fernández 2015; Moldovan and Abbeel 2012). The safety fragment of temporal logic that we consider is more general than the notion of safety of García and Fernández (which is technically a so-called *invariance property* (Baier and Katoen 2008)). One way of guiding exploration in learning is to provide *teacher advice*. A teacher (usually a human) provides advice (e.g., safe actions) when either the learner (Pecka and Svoboda 2014; Clouse 1997) or the teacher (Vidal et al. 2013; Thomaz and Breazeal 2006) considers it to be necessary to prevent catastrophic situations. For example, in a Q-learning setting, the agent acts on the teacher's advice, whenever advice is provided. Otherwise, the agent chooses randomly between the set of actions with the highest Q-values. In each time step, the human teacher tunes the reward signal before sending it to the agent (Thomaz and Breazeal 2006; 2008). Our work is closely related to teacher-guided RL, since a shield can be considered as a teacher, who provides safe actions only if absolutely necessary. In contrast to previous work, the reward signal does not have to be manipulated by the shield, since the shield corrects unsafe actions in the learning and deployment phases.

Safety in Formal Methods. Traditional correct-by-construction controller computation techniques are based on computing an abstraction of the environment dynamics and deriving a controller that guarantees to satisfy the specification under the known environment dynamics. Such methods combine *reactive synthesis* with faithful environment modelling and abstraction. Wongpiromsarn et al. (Wongpiromsarn, Topcu, and Murray 2012) define a receding horizon control approach that combines continuous control with discrete correctness guarantees. For simple system dynamics, the controller can be computed directly (Henzinger and Kopke 1999). For more complex dynamics, both approaches are computationally too difficult. A mitigation strategy is to compute a set of low-level motion primitives to be combined to an overall strategy (DeCastro and Kress-Gazit 2016). Having many motion primitives however also leads to inefficiency. All of the above approaches have in common that a faithful, yet precise enough, abstraction of the physical environment is required, which is not only difficult to obtain in practice, but also introduces the mentioned computational burden. Control methods based on RL partly address this problem, but do not typically incorporate any correctness guarantees. Wen et al. (Wen, Ehlers, and Topcu 2015) propose a method to combine strict correctness guarantees with RL. They start with a non-deterministic correct-by-construction strategy and then perform RL to limit it towards cost optimality without having to know the cost function a priori. Unlike the approach in the paper, their technique does not work with function approximation, which prevents it from being used in complex scenarios. Junges et al. (Junges et al. 2016) adopt a similar framework in a stochastic setting. A major difference between the works by Wen et al. and Junges et al. on the one hand and the shielding framework on the other hand is the fact that the computational cost of the construction of the shield depends on the complexity of the specification and a very abstract version of the system, and is independent of the state space components of the system to be controlled that are irrelevant for enforcing the safety specification. Fu et al. (Fu and Topcu 2016) establish connections between temporal-logic-constrained strategy synthesis in Markov decision processes and probably-approximately-correct-type bounds in learning (Valiant 1984). Bloem et al. (Bloem et al. 2015) proposed the idea to synthesize a *shield* that is attached to a system to enforce safety properties at run time. We adopt this idea, and present our own realization of a shield, geared to the needs of the learning setting.

Preliminaries

A **word** is defined to be a finite or infinite sequence of elements from some alphabet Σ. The set of finite words over Σ is denoted by Σ^*, and the set of infinite words over Σ is written as Σ^ω. The union of Σ^* and Σ^ω is denoted by the symbol Σ^∞.

A **probability distribution** over a (finite) set X is a function $\mu : X \to [0,1] \subseteq \mathbb{R}$ with $\sum_{x \in X} \mu(x) = \mu(X) = 1$. The set of all distributions on X is denoted by $Distr(X)$.

A **Markov decision process** (MDP) $\mathcal{M} = (S, s_I, \mathcal{A}, \mathcal{P}, \mathcal{R})$ is a tuple with a finite set S of states, a unique initial state $s_I \in S$, a finite set $\mathcal{A} = \{a_1 \dots a_n\}$ of actions, a *probabilistic transition function* $\mathcal{P} : S \times \mathcal{A} \to Distr(S)$, and an *immediate reward function* $\mathcal{R} : S \times \mathcal{A} \times S \to \mathbb{R}$.

In **reinforcement learning** (RL), an agent must learn a behavior through trial-and-error via interactions with an unknown environment modeled by a MDP $\mathcal{M} = (S, s_I, \mathcal{A}, \mathcal{P}, \mathcal{R})$. Agent and environment interact in discrete time steps. At each step t, the agent receives an observation s_t. It then chooses an action $a_t \in \mathcal{A}$. The environment then moves to a state s_{t+1} with the probability $\mathcal{P}(s_t, a_t, s_{t+1})$ and determines the reward $r_{t+1} = \mathcal{R}(s_t, a_t, s_{t+1})$. We refer to negative rewards $r_t < 0$ as *punishments*. The *return* $R = \sum_{t=0}^{\infty} \gamma^t r_t$ is the cumulative future discounted reward, where r_t is the immediate reward at time step t, and $\gamma \in [0,1]$ is the *discount factor* that controls the influence of future rewards. The objective of the agent is to learn an *optimal policy* $\Pi : S \to \mathcal{A}$ that maximizes (over the class of policies considered by the learner) the expectation of the return; i.e. $max_{\pi \in \Pi} E_\pi(R)$, where $E_\pi(.)$ stands for the ex-

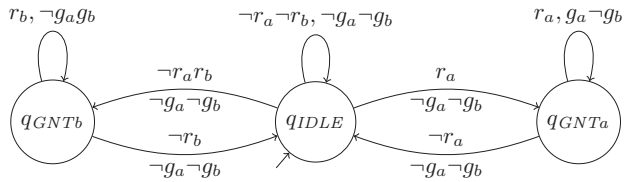

Figure 2: Reactive system implementing a simple arbiter. The transitions of the system are labeled by the input upon reading which causes the transition to be taken and the output selected by the system in this case.

pectation w.r.t. the policy π.

We consider a **reactive system** with a finite set $I = \{i_1, \ldots, i_m\}$ of Boolean input propositions and a finite set $O = \{o_1, \ldots, o_n\}$ of Boolean output propositions. The input alphabet is $\Sigma_I = 2^I$, the output alphabet is $\Sigma_O = 2^O$, and $\Sigma = \Sigma_I \times \Sigma_O$. We refer to words over Σ as *traces* $\overline{\sigma}$. We write $|\overline{\sigma}|$ for the length of a trace $\overline{\sigma} \in \Sigma^\infty$. A set of words $\mathcal{L} \subseteq \Sigma^\infty$ is called a *language*.

As an example, consider a simple arbiter that coordinates the access of some shared resource for two clients. The clients send *requests* for the shared resource to the arbiter with the input propositions $\{r_a, r_a\}$; whenever proposition r_c has a true value, then client $c \in \{a, b\}$ requests access. The arbiter provides *grants* to the clients by indicating them with the values of the output propositions $\{g_a, g_b\}$, where $g_c = \mathsf{true}$ indicates that client c can make use of the shared resource. The values of the propositions change during the trace of the arbiter such that the clients can take turn in utilizing the shared resource.

A **finite-state reactive system** is a tuple $\mathcal{S} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$ with the input alphabet Σ_I, the output alphabet Σ_O, a finite set of states Q, and the initial state $q_0 \in Q$. Function $\delta : Q \times \Sigma_I \to Q$ is a complete transition function, and $\lambda : Q \times \Sigma_I \to \Sigma_O$ is a complete output function. Given the input trace $\overline{\sigma_I} = x_0 x_1 \ldots \in \Sigma_I^\infty$, the system $\mathcal{S}$ produces the output trace $\overline{\sigma_O} = \mathcal{S}(\overline{\sigma_I}) = \lambda(q_0, x_0)\lambda(q_1, x_1) \ldots \in \Sigma_O^\infty$, where $q_{i+1} = \delta(q_i, x_i)$ for all $i \geq 0$. The input and output traces can be merged to the *trace of* $\mathcal{S}$ over the alphabet $\Sigma_I \times \Sigma_O$, which is defined as $\overline{\sigma} = (x_0, \lambda(q_0, x_0))(x_1, \lambda(q_1, x_1)) \ldots \in (\Sigma_I \times \Sigma_O)^\omega$. The type of finite-state reactive system used in this paper is also called *Mealy machines* in the literature.

Fig. 2 shows an example of a reactive system $\mathcal{S} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$ for the resource arbiter introduced above. The system $\mathcal{S}$ has the components $\Sigma_I = \{r_a r_b, \neg r_a r_b, r_a \neg r_b, \neg r_a \neg r_b\}$, $\Sigma_O = \{g_a g_b, \neg g_a g_b, g_a \neg g_b, \neg g_a \neg g_b\}$, $Q = \{q_{IDLE}, q_{GNT0}, q_{GNT1}\}$, the transition function δ (which, e.g., has $\delta(q_{IDLE}, \neg r_a \neg r_b) = q_{IDLE}$), and the output labeling function λ (which, e.g., has $\lambda(q_{IDLE}, \neg r_a \neg r_b) = \neg g_a \neg g_b$). The three states represent which grant was given last during the execution of the system, where the system being in q_{IDLE} denotes that no grant has been given in the previous step of the system's trace.

A **specification** φ defines a set $\mathcal{L}(\varphi) \subseteq \Sigma^\infty$ of allowed traces. The reactive system $\mathcal{S}$ *realizes* φ, denoted by $\mathcal{S} \models \varphi$, iff $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$. φ is *realizable* if there exists such an $\mathcal{S}$. We assume φ is a set of *properties* $\{\varphi_1, \ldots, \varphi_l\}$ such that $\mathcal{L}(\varphi) = \bigcap_i \mathcal{L}(\varphi_i)$. A system satisfies φ iff it satisfies all its properties.

In formal methods, specifications of reactive systems are typically given as formulas in **temporal logic**. *Linear temporal logic* (Pnueli 1977) (LTL) is a commonly used formal specification language. Given a set of propositions AP, an LTL formula describes a language in $(2^{\mathsf{AP}})^\omega$. LTL extends Boolean logic by the introduction of temporal operators such as X (next time), G (globally/always), F (eventually), and U (until). To use LTL for specifying a set of allowed traces by a reactive system, the joint alphabet $\Sigma = \Sigma_I \times \Sigma_O$ of the system must be decomposable into $\Sigma = 2^{\mathsf{AP}_I} \times \Sigma_I^{rest} \times 2^{\mathsf{AP}_O} \times \Sigma_O^{rest}$ for some system input and output components Σ_I^{rest} and Σ_O^{rest} that we do not want to reason about in the LTL specification. Then, the LTL formula can use $\mathsf{AP} = \mathsf{AP}_I \cup \mathsf{AP}_O$ as the set of atomic propositions. Given a trace $\overline{\sigma}$, we write $\overline{\sigma}_{\mathsf{AP}}$ to denote a copy of the trace where, in each character, the factors Σ_O^{rest} and Σ_I^{rest} have been stripped away so that $\overline{\sigma}_{\mathsf{AP}} \in (2^{\mathsf{AP}})^\omega$.

Let us consider an example for an LTL specification (for the arbiter introduced above) that we build from ground up. By default, LTL formulas are evaluated at the first element of a trace. The LTL formula r_a holds on a trace $\overline{\sigma}_{\mathsf{AP}} = \overline{\sigma}_0 \overline{\sigma}_1 \overline{\sigma}_2 \ldots \in (2^{\mathsf{AP}})^\omega$ if and only if $r_a \in \overline{\sigma}_0$. The next-time operator X allows to look one step into the future, so the LTL formula $\mathsf{X}g_a$ holds if $g_a \in \overline{\sigma}_1$. We can take the disjunction between the formulas r_a and $\mathsf{X}g_a$ to obtain an LTL formula $(r_a \vee \mathsf{X}g_a)$ which holds for a trace if at least one of r_a or $\mathsf{X}g_a$ hold. We can then wrap $(r_a \vee \mathsf{X}g_a)$ into the temporal operator G to obtain $\mathsf{G}(r_a \vee \mathsf{X}g_a)$. The effect of adding this operator is that in order for $\overline{\sigma}_{\mathsf{AP}}$ to satisfy $\mathsf{G}(r_a \vee \mathsf{X}g_a)$, the subformula $(r_a \vee \mathsf{X}g_a)$ has to hold at every position in the trace. All in all, we can formalize this description by stating that we have that $\overline{\sigma} \models \mathsf{G}(r_a \vee \mathsf{X}g_a)$ holds if and only if for every $i \in \mathbb{N}$, at least one of $r_a \in \overline{\sigma}_i$ and $g_a \in \overline{\sigma}_{i+1}$ holds. Note that the reactive system given in Figure 2 does not satisfy the specification $\mathsf{G}(r_a \vee \mathsf{X}g_a)$ along all of its traces. The system induces, for instance, a trace of the form $\overline{\sigma} = (\neg r_a \neg r_b, \neg g_a \neg g_b)^\omega$ that results from staying in the q_{IDLE} state forever, along which this specification is not fulfilled. A specification that is however satisfied along all traces of the system is $\mathsf{G}(\mathsf{G}(r_a \wedge \neg r_b) \rightarrow \mathsf{F}\mathsf{G}g_a)$, which can be read as "If from some point onwards, request r_a is always set to true while request proposition r_b is not, then eventually, a grant is given to process a for eternity."

A specification is called a *safety specification* if every trace $\overline{\sigma}$ that is not in the language represented by the specification has a prefix such that all words starting with the prefix are also not in the language. Intuitively, a safety specification states that "something bad should never happen". Safety specifications can be simple *invariance properties* (such as "the level of a water tank should never fall below 1 liter"), but can also also be more complex (such as "whenever a valve is opened, it stays open for at least three seconds"). For specifications in LTL, it is known how to check if it is a safety language and how to compute a *safety automaton*

that represents it (Kupferman and Vardi 2001). Such an automaton is defined as a tuple $\varphi^s = (Q, q_0, \Sigma, \delta, F)$, where $\Sigma = \Sigma_I \times \Sigma_O$, $\delta : Q \times \Sigma \to Q$, and $F \subseteq Q$ is a set of safe states. A *run* induced by a trace $\overline{\sigma} = \sigma_0\sigma_1 \ldots \in \Sigma^\infty$ is a sequence of states $\overline{q} = q_0q_1 \ldots$ such that $q_{i+1} = \delta(q_i, \sigma_i)$ for all $i \in \mathbb{N}$. A trace $\overline{\sigma}$ of a system $\mathcal{S}$ *satisfies* φ^s if the induced run visits only safe states, i.e., $\forall i \geq 0 \,.\, q_i \in F$. The next section gives a detailed example of a safety specification.

A (2-player, alternating) **game** is a tuple $\mathcal{G} = (G, g_0, \Sigma_I, \Sigma_O, \delta, \mathsf{win})$, where G is a finite set of game states, $g_0 \in G$ is the initial state, $\delta : G \times \Sigma_I \times \Sigma_O \to G$ is a complete transition function, and $\mathsf{win} : G^\omega \to \mathbb{B}$ is a winning condition. The game is played by the system and the environment. In every state $g \in G$ (starting with g_0), the environment chooses an input $\sigma_I \in \Sigma_I$, and then the system chooses some output $\sigma_O \in \Sigma_O$. These choices by the system and the environment define the next state $g' = \delta(g, \sigma_I, \sigma_O)$, and so on. The resulting (infinite) sequence $\overline{g} = g_0g_1 \ldots$ is called a *play*. A play is *won* by the system iff $\mathsf{win}(\overline{g})$ is true. A (memoryless) *strategy* for the system is a function $\rho : G \times \Sigma_I \to \Sigma_O$. A strategy is *winning* for the system if all plays $\overline{g}$ that can be constructed when defining the outputs using the strategy (for the respective previous state in the play and the previous environment player move) satisfy $\mathsf{win}(\overline{g})$. The *winning region* W is the set of states from which a winning strategy exists.

A **safety game** defines win via a set $F^g \subseteq G$ of safe states: $\mathsf{win}(g_0g_1 \ldots)$ is true iff $\forall i \geq 0 \,.\, g_i \in F^g$, i.e., if only safe states are visited.

Safety, Abstractions, and Games

The goal of this paper is to combine the best of two worlds, namely (1) the formal correctness guarantees of a controller w.r.t. a temporal logic specification, as provided by formal methods (and reactive synthesis in particular), and (2) the optimality w.r.t. an a priori unknown performance criterion, as provided by RL.

Consider the example of a path planner for autonomous vehicles. Many requirements on system behaviors such as safety concerns may be known and expressed as specifications in temporal logic and can be enforced by reactive controllers. This includes always driving in the correct lane, never jumping the red light, and never exceeding the speed limit (Wen, Ehlers, and Topcu 2015). A learning algorithm is able to incorporate more subtle considerations, such as specific intentions for the current scenario and personal preferences of the human driver, such as reaching some goal quickly but at the same time driving smoothly. By combining RL with reactive synthesis, we achieve *safe reinforcement learning*, which we define in the following way:

Definition 1 *Safe RL is the process of learning an optimal policy while satisfying a temporal logic safety specification φ^s during the learning and execution phases.*

In the following, we consider a safety specification to be given in the form of a deterministic safety word automaton $\varphi^s = (Q, q_0, \Sigma, \delta, F)$, i.e., an automaton in which only safe states in F may be visited.

Reactive synthesis enforces φ^s by solving a *safety game* built from φ^s and an abstraction of the environment in which the policy is to be executed. The game is played by the environment and the system. In every state $q \in Q$, the environment chooses an input $\sigma_I \in \Sigma_I$, and then the system chooses some output $\sigma_O \in \Sigma_O$. The play is won by the system if only safe states in F are visited during the play. In order to win, the system has to plan ahead: it can never allow the play to visit a state from which the environment can force the play to visit an unsafe state in the future.

Planning ahead is the true power of synthesis. Let us revisit the autonomous driver example. Suppose that the car is heading towards a cliff. In order to enforce that the car never crosses the cliff, it has to be slowed down long before it reaches the cliff, and thus far before an abnormal operating condition such as falling down can possibly be detected. In particular, the system has to avoid all states from which avoiding to reach the cliff is no longer possible.

Planning ahead does not require the environment dynamics to be completely known in advance. However, to reason about when exactly a specification violation cannot be avoided, we have to give a (coarse finite-state) abstraction of the environment dynamics. Given that the environment is often represented as an MDP, such an abstraction has to be conservative w.r.t. the behavior of the real MDP. This approximation may have finitely many states even if the MDP has infinitely many states and/or is only approximately known.

Formally, given an MDP $\mathcal{M} = (S, s_I, \mathcal{A}, \mathcal{P}, \mathcal{R})$ and an *MDP observer function* $f : S \to L$ for some set L, we call a deterministic safety word automaton $\varphi^{\mathcal{M}} = (Q, q_0, \Sigma, \delta, F)$ an *abstraction* of $\mathcal{M}$ if $\Sigma = \mathcal{A} \times L$ and for every trace $s_0s_1s_2 \ldots \in S^\omega$ with the corresponding action sequence $a_0a_1 \ldots \in \mathcal{A}^\omega$ of the MDP, for every automaton run $\overline{q} = q_0q_1 \ldots \in Q^\omega$ of $\varphi^{\mathcal{M}}$ with $q_{i+1} = \delta(q_i, (l_i, a_i))$ for $l_i = L(s_i)$ and all $i \in \mathbb{N}$, we have that $\overline{q}$ always stays in F. An abstraction of an MDP describes how its executions can possibly evolve, and provides the needed information about the environment to allow planning ahead w.r.t. the safety properties of interest. Without loss of generality, we assume that $\varphi^{\mathcal{M}}$ has no states in F from which all infinite paths eventually leave F. This ensures that paths that model traces that cannot occur in $\mathcal{M}$ are rejected by $\varphi^{\mathcal{M}}$ as early as possible.

Example 1 *We want to learn an energy-efficient controller for a hot water storage tank. Stored water is kept warm by a heater whose energy consumption depends on the filling level of the tank, but we do not know what the exact relationship is. The outflow is always between 0 and 1 liters per second, and the inflow is known to be between 1 and 2 liters per second whenever the valve is open (and it is 0 otherwise). The capacity of the tank is limited to 100 liters, and whenever the inflow is switched on or off, the setting has to be kept for at least three seconds to limit the wear-out of the valve. Also, the tank must never overflow or run dry.*

We can express the safety specification for the water tank valve controller using the following linear temporal logic formula: $\mathsf{G}(level > 0) \wedge \mathsf{G}(level < 100) \wedge \mathsf{G}((open \wedge \mathsf{X}close) \rightarrow \mathsf{XX}close \wedge \mathsf{XXX}close) \wedge \mathsf{G}((close \wedge \mathsf{X}open) \rightarrow \mathsf{XX}open \wedge \mathsf{XXX}open)$. *The specification consists of four con-*

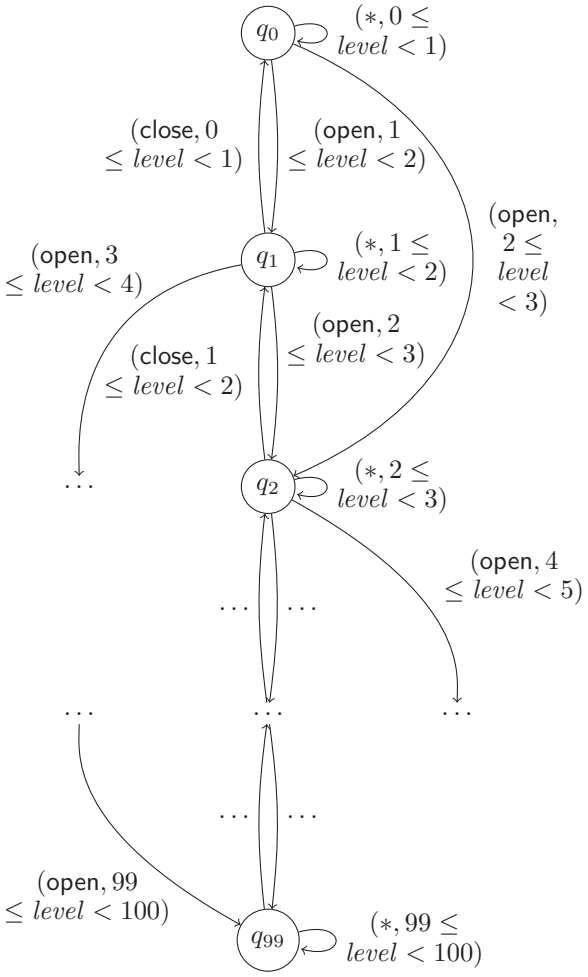

Figure 3: The abstraction of the water tank behavior. All states are accepting and transitions leading to the error state (which exists in addition to the states in the figure and is not accepting) are not shown.

juncts, where the first two conjuncts enforce the water levels to be between the minimum and maximum thresholds. The next conjunct enforces that if the valve is open and then closed, then it has to stay closed for two more time steps (seconds). The final conjunct enforces that if the valve is closed and then opened, it has to stay open for two more time steps.

We can translate the specification to the safety automaton shown in Fig. 4. All states are accepting and transitions leading to the error state (which exists in addition to the states in Fig. 4 and is not accepting) are not shown. It uses the action sets $\mathcal{A} = \{\mathsf{open}, \mathsf{closed}\}$ *for the inflow valve state, and the label set* $L = \{level < 1, 1 \leq level \leq 99, level > 99\}$ *as needed information about the water tank filling status. What we know about the behavior of the water tank can be summarized as the abstraction automaton that we give in Fig. 3.*

The shield that the approach presented in this paper computes ensures that when the water level in the tank becomes too low, the inflow valve is opened until some minimum level of 4 *is reached, and it also prevents the inflow valve from being opened when the level is above* 93*. The latter is necessary as the valve has to stay open for at least three time steps. So as the inflow may be up to 2 liters/second during this time and the outflow may be 0, there is otherwise an overflow risk. As the shield is generated using the specification, it plans ahead for this not to happen, so it must prevent the opening of the inflow valve if the level is above* 93*.*

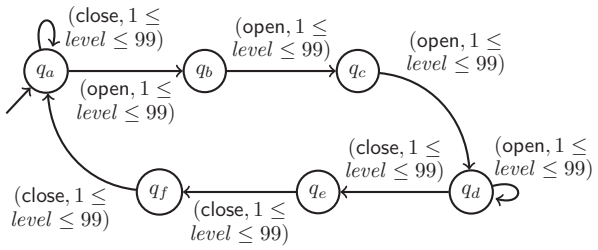

Figure 4: The specification for the water tank controller.

Shielded Reinforcement Learning

We introduce a *correct-by-construction* reactive system, called a shield, into the traditional learning process. We make the following assumptions.

Assumption 1 *(i) The environment can be modeled as an MDP* $\mathcal{M} = (S, s_I, \mathcal{A}, \mathcal{P}, \mathcal{R})$*. (ii) We have constructed an abstraction* $\varphi^{\mathcal{M}}$*. (iii) The learner accepts elements from* $S \times Q$ *as state input (for the state space of the shield* Q*).*

We describe the operation of a learner and a shield together in this section, and give the construction for computing the shield in the next section. The shield will be given as a reactive system $\mathcal{S} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$.

The shield monitors the actions of the agent, and substitutes the selected actions by safe actions whenever this is necessary to prevent the violation of φ^s. In each step t, the agent selects an action a_t^1. The shield forwards a_t^1 to the environment, i.e., $a_t = a_t^1$. Only if a_t^1 is unsafe w.r.t. φ^s, the shield selects a different action $a_t \neq a_t^1$ instead. The environment executes a_t, moves to s_{t+1} and provides r_{t+1}. The agent receives a_t and r_{t+1}, and updates its policy for a_t using r_{t+1}. The question is what the reward for a_t^1 should be in case we have $a_t \neq a_t^1$. We discuss two different approaches.

1. **Assign a punishment r'_{t+1} to a_t^1.** The agent assigns a punishment $r'_{t+1} < 0$ to a_t^1 and learns that selecting a_t^1 at state s_t is unsafe, without ever violating φ^s. However, there is no guarantee that unsafe actions are not part of the final policy. Therefore, the shield has to remain active even after the learning phase.
2. **Assign the reward r_{t+1} to a_t^1.** The agent updates a_t^1 with the reward r_{t+1}. Therefore, picking unsafe actions can likely be part of an optimal policy by the agent. Since an unsafe action is always mapped to a safe one, this does not pose a problem and the agent never has to learn to avoid

unsafe actions. Consequently, the shield is (again) needed during the learning and execution phases.

In order to be less restrictive to the learning algorithm, we propose that in every time step, the agent provides a ranking $rank_t = (a_t^1, \ldots, a_t^k)$ on the allowed actions, i.e., the agent wants a_t^1 to be executed the most, a_t^2 to be executed the second most, etc. The ranking does not have to contain all available actions, i.e. $1 \leq |rank_t| \leq n_t$, where n_t is the number of available actions in step t. The shield selects the first action $a_t \in rank_t$ that is safe according to φ^s. Only if all actions in $rank_t$ are unsafe, the shield selects a safe action $a_t \notin rank_t$. Both approaches for updating the policy discussed before can naturally be extended for a ranking of several actions. A second advantage of having a ranking on actions is that the agent can perform several policy updates at once; e.g., if all actions in $rank_t$ are unsafe, the agent can perform $|rank_t| + 1$ policy updates in one step by using the rewards r'_{t+1} or r_{t+1} for all of them, depending on which of the above variants is used.

Synthesis of Shields

A shield $\mathcal{S}$ enforces two properties: *correctness* and *minimum interference*. First, $\mathcal{S}$ enforces correctness against a given safety specification φ^s.With minimum interference, we mean that the shield restricts the agent as rarely as possible. $\mathcal{S}$ is computed by reactive synthesis from φ^s and an MDP abstraction $\varphi^{\mathcal{M}}$ that represents the environment in which the agent shall operate.

We give an algorithm to compute shields. We establish (in the next section) that the computed shields (1) enforce the correctness criterion, and (2) are the minimally interfering shields among those that enforce φ^s on all MDPs for which $\varphi^{\mathcal{M}}$ is an abstraction.

Given is an RL problem in which an agent has to learn an optimal policy for an unknown environment that can be modelled by an MDP $\mathcal{M} = (S, s_I, \mathcal{A}, \mathcal{P}, \mathcal{R})$ while satisfying $\varphi^s = (Q, q_0, \Sigma, \delta, F)$ with $\Sigma = \Sigma_I \times \Sigma_O$ and $\mathcal{A} = \Sigma_O$. We assume some abstraction $\varphi^{\mathcal{M}} = (Q_{\mathcal{M}}, q_{0,\mathcal{M}}, \mathcal{A} \times L, \delta_{\mathcal{M}}, F_{\mathcal{M}})$ of $\mathcal{M}$ for some MDP observer function $f : S \to L$ to be given. Since φ^s models a restriction of the traces of the MDP and the learner together that we want to enforce, we assume it to have $\Sigma = L \times \mathcal{A}$, i.e., it reads the part of the system behavior that the abstraction is concerned with. We perform the following steps.

1. We translate φ^s and $\varphi^{\mathcal{M}}$ to a safety game $\mathcal{G} = (G, g_0, \Sigma_I, \Sigma_O, \delta, F^g)$ between two players. In the game, the environment player chooses the next observations from the MDP state (i.e., elements from L), and the system chooses the next action. Formally, $\mathcal{G}$ has the following components: $G = Q \times Q_{\mathcal{M}}, g_0 = (q_0, q_{0,\mathcal{M}}), \Sigma_I = L, \Sigma_O = \mathcal{A}, \delta((q, q_{\mathcal{M}}), l, a) = (\delta(q, (l, a)), \delta_{\mathcal{M}}(q, (l, a)))$, for all $(q, q_{\mathcal{M}}) \in G, l \in L, a \in \mathcal{A}$, and $F^g = (F \times Q_{\mathcal{M}}) \cup (Q \times (Q_{\mathcal{M}} \setminus F_{\mathcal{M}}))$.
2. Next, we compute the winning region $W \subseteq F^g$ of G as described by Bloem et al. (2015).
3. We translate G and W to a reactive system $\mathcal{S} = (Q_{\mathcal{S}}, q_{0,\mathcal{S}}, \Sigma_{I,\mathcal{S}}, \Sigma_{O,\mathcal{S}}, \delta_{\mathcal{S}}, \lambda_{\mathcal{S}})$ that constitutes the shield.

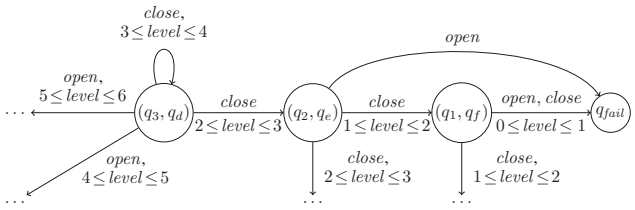

Figure 5: An excerpt for the product game of our running example. Transitions to paradise states are not shown.

The shield has the following components: $Q_{\mathcal{S}} = G, q_{0,\mathcal{S}} = (q_0, q_{0,\mathcal{M}}), \Sigma_{I,\mathcal{S}} = L \times \mathcal{A}, \Sigma_{O,\mathcal{S}} = \mathcal{A}$, $\delta_{\mathcal{S}}(g, (l, a)) = \delta(g, (l, \lambda_{\mathcal{S}}(g, (l, a))))$ for all $g \in G, l \in L, a \in \mathcal{A}$, and

$$\lambda_{\mathcal{S}}(g, l, a) = \begin{cases} a & \text{if } \delta(g, (l, a)) \in W \\ a' & \text{if } \delta(g, (l, a)) \notin W \text{ for some arbitrary} \\ & \text{but fixed } a' \text{ with } \delta(g, (l, a')) \in W. \end{cases}$$

In the construction, the state space of the game is the product between the specification automaton state set and the abstraction state set. The safe states in the game (in the set F^g) are the ones at which either the specification automaton is in a safe state, or the abstraction is in an unsafe state. The latter case represents that the observed MDP behavior differs from the behavior that was modeled in the abstraction. For game solving, it is important that such cases (whose occurrence in the field witnesses the incorrectness of the abstraction) count as winning for the system player, as the system player only needs to play correctly in environments that conform to the abstraction.

To exemplify the shield construction, let us reconsider Example 1. Building the product game between the specification automaton and the MDP abstraction (that we detail in the next section) leads to a game with 602 states (if we merge all states in $F \times Q_{\mathcal{M}}$ into a single error state and all states in $Q \times (Q_{\mathcal{M}} \setminus F_{\mathcal{M}})$ into a single *paradise state* from which the game is always won by the system). If we solve the game, then most of the states are winning, but a few are not. Fig. 5 shows a small fraction of the game that contains such non-winning states. In state (q_3, q_d), the system should not choose action $close$, as otherwise the system cannot avoid to reach q_{fail} for some possible evolution of the environment that is consistent with our abstraction. It could be the case that q_{fail} is not reached when the environment chooses to let the level stay the same for a step, but the system cannot be sure about this, so the action $close$ must not be picked.

A shield allows all actions that are guaranteed to lead to a state in W, no matter what the next observation is. Since these states, by the definition of the set of winning states, are exactly the ones from which the system player can enforce not to ever visit a state not in F, the shield is minimally interfering. It disables all actions that may lead to an error state (according to the abstraction).

The construction of a shield can be extended naturally if a ranking of actions $rank_t = \{a_t^1, \ldots, a_t^n\}$ is provided by the agent. Then, the shield selects the first action $a_t = a_t^i$ that is allowed by φ^s. Only if all actions in $rank_t$ are unsafe, the shield is allowed to deviate and to select a safe action $a_t \notin rank_t$.

Correctness and Minimal Interference of the Computed Shields

We now prove that the shields computed according to the definitions indeed have the claimed properties, namely *correctness*, and *minimal interference*.

Correctness: A shield works correctly if for every trace $s_0 a_0 s_1 a_1 \ldots \in (S \times \mathcal{A})^\omega$ that MDP, shield and learner can together produce, we have that $(f(s_0), a_0)(f(s_1), a_1)\ldots$ is in the language of the specification automaton φ^S for the MDP labeling function f. Additionally, the shield must always report at least one available action at every step.

Let $q_0 q_1 \ldots \in Q^\omega$ be the run of $\mathcal{S}$ corresponding to a trace $s_0 a_0 s_1 a_1 \ldots$ of the abstraction, i.e., for which for every $i \in \mathbb{N}$, we have $a_i = \lambda_{\mathcal{S}}(q_i, (f(s_i), a_i))$ and $q_{i+1} = \delta_{\mathcal{S}}(q_i, (f(s_i), a_i))$. By the construction of the shield, we have that $Q_{\mathcal{S}} = Q \times Q_{\mathcal{M}}$, where Q is the state space of φ^S and $Q_{\mathcal{M}}$ is the state space of the abstraction. Hence, we can also write $q_0 q_1 \ldots$ as $(q_0^S, q_0^{\mathcal{M}})(q_1^S, q_1^{\mathcal{M}})\ldots$, where $q_0^{\mathcal{M}} q_1^{\mathcal{M}} \ldots$ is the run of the abstraction automaton on $s_0 a_0 s_1 a_1 \ldots$ and $q_0^S q_1^S \ldots$ is a run of φ^S on $s_0 a_0 s_1 a_1 \ldots$. By the construction of the shield, it only has reachable states $(q^S, q^{\mathcal{M}})$ that are in the set of winning positions. For all such states and all possible next labels $l \in L$, there exists at least one action such that if the action is taken, then the next state $(q'^S, q'^{\mathcal{M}})$ is winning as well. The q^S component of the run of the shield always reflects the state of the specification automaton along the trace, and since a winning strategy makes sure that only winning states are ever visited along a play, by the definition of F^g, the error state of φ^S can only be visited after the error state for the abstraction MDP has been visited (and hence the abstraction turned out to be incorrect). Note that actions that may lead to the violation of the specification (according to the abstraction) are modified by the shield whenever the prefix trace is accepted by the abstraction, as otherwise a state that is not winning in the game would be visited, which the definition of $\lambda_{\mathcal{S}}$ prevents.

Minimal Interference: Let the shield, learner, and MDP together produce a prefix trace $s_0 a_0 s_1 a_1 s_2 a_2 \ldots s_k$ that induces a (prefix) run $q_0 q_1 \ldots q_{k-1} \in Q^*$ of the safety automaton φ^S that we used as the representation of the specification for building the shield. Assume that the shield deactivates an action a_{k+1} that is available from state s_k in the MDP. We show that the shield had to deactivate a_{k+1} as there is another MDP that is consistent with the observed behavior and the abstraction for which, regardless of the learner's policy, there is a non-zero probability to violate the specification after the trace prefix $s_0 a_0 s_1 a_1 s_2 a_2 \ldots s_k a_{k+1}$.

Using the abstract finite-state machine $\varphi^{\mathcal{M}} = (Q_{\mathcal{M}}, q_{0,\mathcal{M}}, \Sigma, \delta, F)$, we define this other MDP $\mathcal{M}' = (S', s'_I, \mathcal{A}, \mathcal{P}', \mathcal{R})$ with $S' = Q_{\mathcal{M}} \times L$, $s'_I = (q_{0,\mathcal{M}}, f(s_0))$, $\mathcal{A}$ being the same set of actions as in the original MDP, and where $\mathcal{P}'((q,l),a)$ is a uniform distribution over all elements from the set $\{(q', l') \in Q_{\mathcal{M}} \times L \mid q' = \delta(q, (l, a)), q' \in F, \exists a' \in \mathcal{A}.\delta(q', (l', a')) \in F\}$ for every $(q, l) \in S'$ and $a \in \mathcal{A}$. Every state $(q', l') \in S'$ is mapped to l' by the abstraction function f. The reward function is the same as in the original MDP, except that we ignore the (new) state component of the shield.

Assume now that action a_{k+1} is activated after the prefix trace $s_0 a_0 s_1 a_1 s_2 a_2 \ldots s_k$ while the shield is in a state $(q^{\mathcal{S}}, q^{\mathcal{M}})$. We have that $\mathcal{M}'$ is an MDP in which every finite-length label sequence that is possible in the abstraction for some action sequence has a non-zero probability to occur if the action sequence is chosen. Due to the construction of the shield by game solving, action a_{k+1} is only deactivated in state $(q^{\mathcal{S}}, q^{\mathcal{M}})$ if in the game, the environment player had a strategy to violate φ^S using only traces allowed by the abstraction. Since φ^S is a safety property, the violation would occur in finite time. Since in $\mathcal{M}'$, all finite traces that can occur in the abstraction have a non-zero probability, activating a_{k+1} (and the learner choosing a_{k+1}) would imply a non-zero proability to violate the specification in the future, no matter what the learner does in the future. Hence, the shield could not prevent a violation in such a case, and a_{k+1} needs to be deactivated.

Convergence

Define an MDP $\mathcal{M} = (S, s_I, \mathcal{A}, \mathcal{P}, \mathcal{R})$, with discrete state set S, discrete state-dependent action sets $\mathcal{A}_s$, and state-dependent transition functions $\mathcal{P}_s(a, s')$ that define the probability of transitioning to state s' when taking action a in state s. Assume also that a shield $\mathcal{S} = (Q_{\mathcal{S}}, q_{0,\mathcal{S}}, \Sigma_{I,\mathcal{S}}, \Sigma_{O,\mathcal{S}}, \delta_{\mathcal{S}}, \lambda_{\mathcal{S}})$ is given for $\mathcal{M}$ and for some MDP labeling function $f : S \to L$.

We can build a product MDP $\mathcal{M}'$ that represents the behavior of the shield and the MDP together. Since $\mathcal{M}'$ is a standard MDP, all learning algorithms that converge on standard MDPs can be shown to converge in the presence of a shield under this construction. Note that this argument requires that whenever an action ranking is chosen by the learner that does not contain a safe action, there is a fixed probability distribution over the safe actions executed instead. This distribution may depend on the state of the MDP and the shield and the selected ranking, but must be constant over time, as otherwise we could not model the joint behavior of the shield and the environment MDP as a product MDP. Note that we made use of the fact that the learner has access to the state of the shield and can base its actions on it in this argument. In such a case, it suffices for the learner to observe the current state as state of $\mathcal{M}$ rather than $\mathcal{M}'$. To the learner, this is indistinguishable from operating on $\mathcal{M}$ without a shield.

Experiments

We applied shielded RL in four domains: (1) a robot in a grid world, (2) a self-driving car scenario, (3) the water tank scenario from Example 1, and (4) the pacman example. For

clarity, we compare between a subset of shielding settings which we later specify for each problem. The simulations were performed on a computer equipped with an Intel® Core™i7-4790K and 16 GB of RAM running a 64-bit version of Ubuntu® 16.04 LTS. Source code, input files, and detailed instructions to reproduce our experiments are available for download.[1]

Grid world Example. We conduced two experiments: a robot in a 9x9 grid world, and a robot in a 15x9 grid world with a moving obstacle. In both experiments, the robot's objective is to visit all the colored regions in a given order while maintaining the following safety property φ^s: the robot must not crash into walls or the moving opponent agent. If the robot visits all marked regions in a given order (called episode), a reward is granted, and if φ^s is violated, a penalty is applied.

The agent uses tabular Q-learning with an ϵ-greedy explorer that is capable of multiple policy updates at once. For both settings, we synthesized a shield from φ^s and the (precise) environment abstraction in less than 2 seconds.

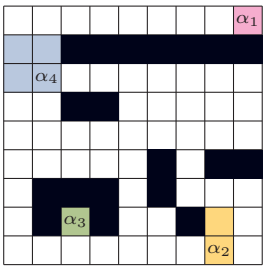

Figure 6: 9x9 grid world.

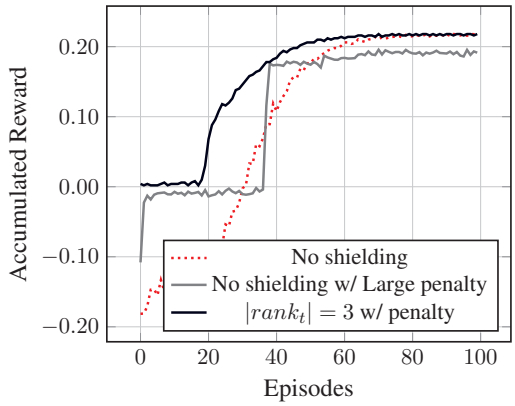

Figure 7: The accumulated reward per episode for the 9x9 grid world example.

The 9x9 grid world of the first experiment is illustrated in Fig. 6. The result is shown in Fig. 7. We compare between no shielding (red, dashed), no shielding with large penalties for unsafe actions (black, solid), and a $|rank_t| = 3$ shielding with penalties for corrected actions (black, solid). The results show that only the unshielded versions experience negative rewards. Furthermore, the shielded version is not only safe, but also tends to learn more rapidly.

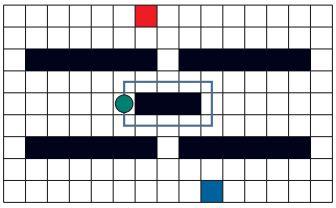

Figure 8: 15x9 grid world with moving obstacle.

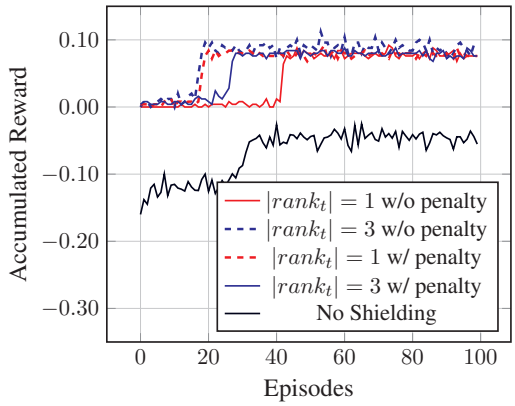

Figure 9: The accumulated reward per episode for the 15x9 grid world example.

The 15x9 grid world of the second experiment is illustrated in Fig. 8. The result is shown in Fig. 9. Only the shielded version with $|rank_t| = 3$ without penalty (blue, dashed) finds the optimal path, resulting in a higher average reward. In scenarios with $|rank_t| = 1$ (red) or with penalties (solid), the agent computes a suboptimal path.

A Self-Driving Car Example. This example considers an agent that learns to drive around a block in a clockwise direction in an environment with the size of 480x480 pixels. The car has 8 sensors distributed evenly around the car that trigger whenever the agent is less than 60 pixels away from a wall. In each step, the car moves 3 pixels in the direction of its heading and can make a maximum turn of 7.5 degrees in either direction. The safety specification in this example is to avoid crashing into a wall. A corresponding shield was synthesized in 2 seconds. In each step, a positive reward is given if the car moves a step in a clockwise direction along its track and a penalty is given if it moves in a counter-clockwise direction. A crash into the wall results in a penalty and a restart. The agent uses a Deep Q-Network with a Boltzmann exploration policy. This network consists of 4 input nodes, 8 outputs nodes and 3 hidden layers.

[1] https://github.com/safe-rl/safe-rl-shielding

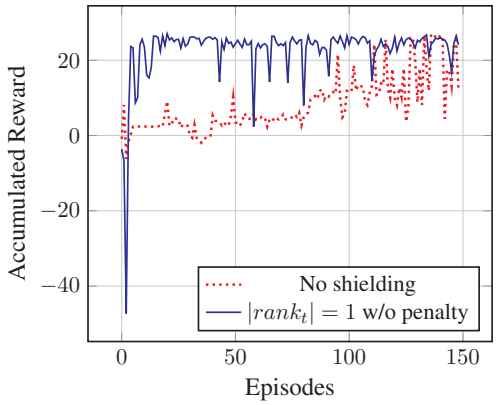

Figure 10: The accumulated reward per episode for the self driving car example.

The plot in Fig. 10 shows that the accumulated rewards for unshielded RL (red, dashed) increases over time, but the car still crashes at the end of the simulation. The shielded version without punishment (blue, solid) learns more rapidly than the unshielded learner and never crashes.

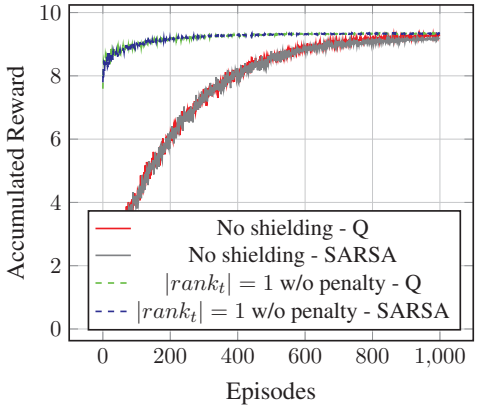

Figure 11: The accumulated reward per episode for the water tank example.

The Water Tank Example. In this example, the tank must never run dry or overflow by controlling the inflow switch (φ_1^s). In addition, the inflow switch must not change its mode of operation before 3 time steps have passed since the last mode change (φ_2^s). Refer to Ex. 1, for a full description of the abstract water tank dynamics and specification. We generated a concrete MDP in which the energy consumption depends only on the state and there are multiple local minima. A shield was synthesized in less than a second. Fig. 11 shows that both shielded (dashed lines) and unshielded Q-learning and SARSA experiments (solid lines) do reach an optimal policy. However, the shielded implementations reach the optimal policy in a significantly shorter time than the unshielded implementations.

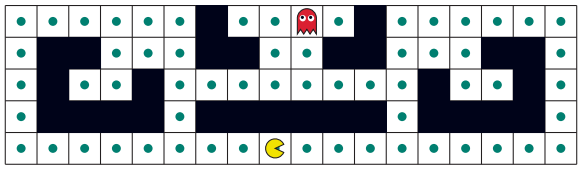

Figure 12: The 5x18 grid world of the pacman example.

The Pacman Example. We conduced another experiment in a 18x5 grid world that simulates a pacman-like environment, as illustrated in Fig. 12. The objective of the robot (pacman) is to visit all fields of the maze without crashing into the moving opponent agent (ghost). If pacman visits all fields in the maze, a reward is granted, and if pacman and the ghost collide, a penalty is applied. The example uses an approximate Q-learning agent.

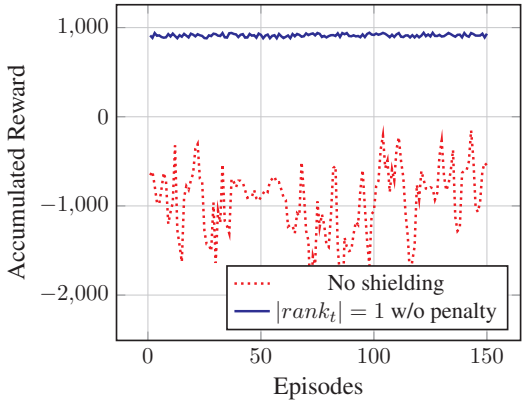

Figure 13: The accumulated reward per episode for the pacman example.

We synthesized a shield in less than 7 seconds. Fig. 13 shows the results. As expected, in the shielded case, pacman is always able to clear the game starting from the first episode.

Conclusion

We developed a method for RL under safety constraints expressed as temporal logic specifications. The method is based on shielding the decisions of the underlying learning algorithm from violating the specification. We proposed an algorithm for the automated synthesis of shields for given temporal logic specifications. Even though the inner working of a learning algorithm is often complex, the safety criteria may still be enforced by possibly simple means. Shielding exploits this possibility. A shield depends only on the monitored input-output behavior, the environment abstraction, and the correctness specifications – it is independent of the intricate details of the underlying learning algorithm.

We demonstrated the use of shielded learning on several RL scenarios. In all of them, the learning performance of the shielded agents improved compared to the unshielded case. The main downside of our approach is that in order to prevent the learner from making unsafe actions, some ap-

proximate model of when which action is unsafe needs to be available. We argue that this is unavoidable if the allowed actions depend on the state of the environment, as otherwise there is no way to know which actions are allowed. Our experiments show, however, that in applications in which safe learning is needed, the effort to construct an abstraction is well-spent, as our approach not only makes learning safe, but also shows great promise of improving learning performance.

Acknowledgments

R. Ehlers was supported by the Institutional Strategy of the University of Bremen, funded by the German Excellence Initiative. R. Bloem and B. Könighofer were supported by the Austrian Science Fund (FWF) through the projects RiSE (S11406-N23) and LogiCS (W1255-N23). The authors U. Topcu and M. Alshiekh were supported partly by the grants AFRL FA8650-15-C-2546, NSF 1652113 and 1646522, DARPA W911NF-16-1-0001 and ARO W911NF-15-1-0592. The authors U. Topcu and S. Niekum were supported partly by NSF 1617639.

References

Baier, C., and Katoen, J.-P. 2008. *Principles of Model Checking*. The MIT Press.

Bloem, R.; Könighofer, B.; Könighofer, R.; and Wang, C. 2015. Shield synthesis: - runtime enforcement for reactive systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st Int. Conf, TACAS 2015*, 533–548.

Clouse, J. 1997. On integrating apprentice learning and reinforcement learning title2:. Technical report, Amherst, MA, USA.

DeCastro, J. A., and Kress-Gazit, H. 2016. Nonlinear controller synthesis and automatic workspace partitioning for reactive high-level behaviors. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016*, 225–234.

Emerson, E. A. 1990. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science (Vol. B)*. Cambridge, MA, USA: MIT Press. chapter Temporal and Modal Logic, 995–1072.

Fu, J., and Topcu, U. 2016. Synthesis of shared autonomy policies with temporal logic specifications. *IEEE Trans. Automation Science and Engineering* 13(1):7–17.

García, J., and Fernández, F. 2015. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research* 16:1437–1480.

Henzinger, T. A., and Kopke, P. W. 1999. Discrete-time control for rectangular hybrid automata. *Theor. Comput. Sci.* 221(1-2):369–392.

Junges, S.; Jansen, N.; Dehnert, C.; Topcu, U.; and Katoen, J. 2016. Safety-constrained reinforcement learning for mdps. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd Int. Conference, TACAS 2016*, 130–146.

Kupferman, O., and Vardi, M. Y. 2001. Model checking of safety properties. *Formal Methods in System Design* 19(3):291–314.

Manna, Z., and Pnueli, A. 1995. *Temporal verification of reactive systems - safety*. Springer.

Moldovan, T. M., and Abbeel, P. 2012. Safe exploration in Markov decision processes. *arXiv preprint arXiv:1205.4810*.

Pecka, M., and Svoboda, T. 2014. *Safe Exploration Techniques for Reinforcement Learning – An Overview*. Cham: Springer International Publishing. 357–375.

Pnueli, A. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, FOCS 1977*, 46–57.

Sutton, R. S., and Barto, A. G. 1998. Reinforcement learning: An introduction. *IEEE Trans. Neural Networks* 9(5):1054–1054.

Thomaz, A. L., and Breazeal, C. 2006. Reinforcement learning with human teachers: Evidence of feedback and guidance with implications for learning performance. In *Proceedings of the 21st National Conference on Artificial Intelligence - Vol. 1*, 1000–1005. AAAI Press.

Thomaz, A. L., and Breazeal, C. 2008. Teachable robots: Understanding human teaching behavior to build more effective robot learners. *Artif. Intell.* 172(6-7):716–737.

Valiant, L. G. 1984. A theory of the learnable. *Commun. ACM* 27(11):1134–1142.

Vidal, P. Q.; Rodríguez, R. I.; González, M. R.; and Regueiro, C. V. 2013. Learning on real robots from experience and simple user feedback. *Journal of Physical Agents* 7(1).

Wen, M.; Ehlers, R.; and Topcu, U. 2015. Correct-by-synthesis reinforcement learning with temporal logic constraints. In *IROS 2015*, 4983–4990.

Wongpiromsarn, T.; Topcu, U.; and Murray, R. M. 2012. Receding horizon temporal logic planning. *IEEE Trans. Automat. Contr.* 57(11):2817–2830.