

Evaluating Probabilistic Model Checking Tools for Verification of Robot Control Policies

Shashank Pathak^{1,2}, Luca Pulina³, and Armando Tacchella²

¹ RBCS, Istituto Italiano di Tecnologia (IIT), Via Morego, 30 – 16163 Genova – Italy
`Shashank.Pathak@iit.it`

² DIBRIS, Università di Genova, Via Opera Pia, 13 – 16145 Genova – Italy
`Armando.Tacchella@unige.it`

³ POLCOMING, Università di Sassari, Viale Mancini 5 – 07100 Sassari – Italy
`lpulina@uniss.it`

Abstract. In the last decade, thanks to the ability of analyzing probabilistic models given specifications in temporal logics, Probabilistic Model Checking (PMC) has become a widely used methodology in several application domains, e.g., communication protocols, planning and scheduling, and robotics. Moreover, the availability of effective PMC tools enables automated analysis in such complex real-world scenarios.

In this paper we evaluate PMC tools to investigate safety of robots whose control policies are learned by reinforcement, i.e., considering feedback signals received upon acting in unknown environments. Introduced in previous contributions of ours, a new challenging application domain for PMC tools is represented by their usage as back-engines of an automated methodology aimed to verify and repair control policies. We present an evaluation of the current state-of-the-art PMC tools and assess their potential on various case studies, including both real and simulated robots accomplishing navigation, manipulation and reaching tasks.

1 Introduction

In the words of [4], Probabilistic Model Checking (PMC) – see, e.g., [9] – aims to achieve automated analysis of probabilistic models against specifications given in temporal logic. PMC techniques can be used with several types of probabilistic models – e.g., continuous-time Markov chains (CTMCs), discrete-time Markov chains (DTMCs), Markov Decision Processes (MDPs) –, and it has become a widely used methodology in several application domains such as communication protocols [10], planning and scheduling [11], and robotics [13]. Moreover, the availability of effective PMC tools enables automated analysis in such complex real-world scenarios.

In this paper we evaluate PMC tools to investigate safety of robots whose control policies are *learned by reinforcement*. Reinforcement Learning (RL) is

Proceedings of the 21st RCRA workshop on *Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion* (RCRA 2014).
Vienna, Austria, July 17–18, 2014.

one of the most widely adopted paradigms to obtain intelligent behavior from interactive robots — see, e.g., [15]. Its fundamental assumption is that robots are capable of perceiving the current state of the environment, as well as feedback signals that occur each time the robot acts. Feedback can be either an immediate reward or penalty, and the ultimate goal of the robot is to find a course of action which is guaranteed to maximize the expected total reward over time. In this sense, RL can be seen as a way to synthesize control programs when knowledge about the external environment is limited. RL methods have shown robust and efficient learning on a variety of robot-control problems — see, e.g. [3]. However, as mentioned in [6], the asymptotic nature of guarantees about the performance of RL makes it difficult to bound the probability of damaging the controlled robot and/or the environment. This shortcoming of RL limits its applicability in all those cases where autonomous robots should be deployed in close contacts with humans, or in other safety-critical contexts.

An interesting research question is thus how to guarantee that, given a control policy synthesized by RL, such policy will have a very low probability of yielding undesirable behaviours like, e.g., damaging the robot or the environment wherein it operates. In particular, we are interested to apply PMC techniques where the interaction between the robot and the environment, as well as the related safety properties, are modeled in a probabilistic logic framework. Introduced in previous contributions of ours [13], a challenging application domain for PMC tools is represented by their usage as back-engines of an automated methodology aimed to verify and repair control policies.

The rest of the paper is organized as follows. In Section 2 we introduce basic notations and definitions related to RL and probabilistic model checking. In Section 3 we present our methodology based on probabilistic model checking to ensure safety of stochastic policies obtained with RL. The main idea behind our approach is to model the interaction between the robot and the environment as a Markov chain computed at the end of the learning process. Then, considering safety properties expressed in probabilistic computational tree logic (PCTL) [2], the likelihood of ill behaviors is turned into a model checking problem on the Markov chain. Finally, automated repair leverages potential counterexamples obtained during verification in order to fix agents, and make them compliant to the stated safety requirements. In Section 4, we evaluate PMC tools on three case studies, namely *air hockey* – introduced in [5] –, *iCub reaching* and *grid world*. In *air hockey*, we consider a simulated manipulator learning to play defense in the air hockey game, wherein speed profiles exceeding the operating range of the actual manipulator can be generated, thus rising a safety concern about the robot integrity. In *iCub reaching*, the humanoid iCub [12] learns to reach a target object in the presence of nearby objects which must be avoided. In this case, environmental safety is the primary concern. Finally, in *grid world*, we consider an artificial domain where a robot is learning to move on a grid from a given source to a specific target in the presence of “pits”, i.e., cells that cause the robot to fail its mission. Albeit conceptually much simpler than the other two case studies, *grid world* is parametrized over the size of the grid.

2 Preliminaries

The formal framework of Reinforcement Learning (RL) is that of Markov Decision Processes (MDPs) wherein an agent can perceive a set S of distinct *states* of its environment and has a set A of *actions* that it can perform. At each discrete *time step* t , the agent senses state s_t , and performs action a_t . The environment responds by giving the agent a *reward* $r_{t+1} = r(s_t, a_t)$ and by producing a *transition* to the succeeding state $s_{t+1} = \delta(s_t, a_t)$. This process is subdivided in *episodes* consisting of a finite number of time steps. While the functions r and δ may not be known, the founding assumption is that they depend only on the current state and action. The task of the agent is to learn a *policy*, i.e., a mapping from the observed state s_t to the action a_t . A *deterministic* policy $\pi : S \rightarrow A$ suggests specific actions, whereas a *stochastic policy* $\pi : S \times A \rightarrow [0, 1]^4$ returns the probability of executing actions. Intuitively, a (deterministic) policy should achieve the greatest possible cumulative reward over time. We can define the *discounted cumulative reward* $V^\pi(s_0)$ as follows:

$$V^\pi(s_0) = r_1 + \gamma r_2 + \gamma^2 r_3 + \dots = \sum_{k=0}^{\infty} \gamma^k r_{k+1} \quad (1)$$

where π is an arbitrary policy, s_0 is an arbitrary initial state, and the sequence of rewards $r_1, r_2, \dots$ is generated by repeatedly using the policy π to select actions, i.e., $a_t = \pi(s_t)$ for all $t \geq 0$. The *discount factor* $0 \leq \gamma < 1$ is a constant that determines the relative value of delayed versus immediate rewards. The value $V^\pi(s_0)$ can be defined also for stochastic policies, if we take the expectation — under the action’s probability distribution - of the left-hand-side in equation (1). Given any two policies π and π' , we define $\pi \geq \pi'$ to be true exactly when $V^\pi(s) \geq V^{\pi'}(s)$ for all $s \in S$. An *optimal policy* π^* is a policy such that $\pi^* \geq \pi$ for every policy π . While π^* may not be unique, all the optimal policies share the *optimal state-value function* denoted by $V^*(s) = \max_{\pi} V^\pi(s)$ for all $s \in S$. An optimal (deterministic) policy is guaranteed to exist as long as (i) all states are visited infinitely often, (ii) rewards are bounded, and (iii) there exists zero-cost absorbing state(s) for all policies — see, e.g., [16]. However, it is usually not possible to achieve an optimal policy through a finite amount of interactions with the environment, which implies that the practical goal of RL agents is to *approximate* π^* .

In Figure 1, we present our learning setup. The diagram in Figure 1 (left) shows two agents, namely **Simulator** and **Learner**. **Simulator** provides a realistic simulation of a physical agent and its surrounding environment. Notice that, even if the the learned control policy is to be deployed on a real robot, we assume that the learning phase is always performed using a simulator to avoid safety issues in this phase. Simulation is arranged into episodes which start every time **Learner** sends a *start* signal to **Simulator**. Then **Simulator** sends to **Learner** the current state s and receives a corresponding action a . Once a is accomplished, **Simulator**

⁴ $[0, 1]$ denotes a closed sub-interval of $\mathbb{R}$, i.e., every $x \in \mathbb{R}$ such that $0 \leq x \leq 1$.

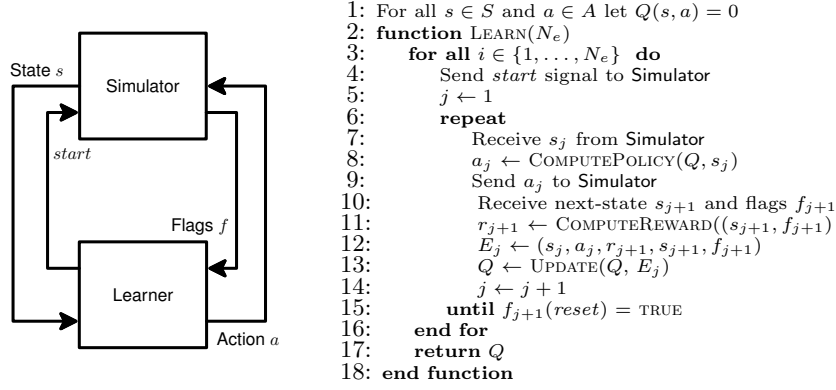


Fig. 1: Functional diagram of a general purpose learning setup (left) and pseudocode of the **Learner** core algorithm (right).

sends back a flag vector f which contains additional feedback. In particular, we assume that the flag $f(\text{reset}) \in \{\text{TRUE}, \text{FALSE}\}$ is always present and, if **TRUE**, it indicates the end of the episode to **Learner**.

Learner approximates an optimal policy π^* which is adequate for the environment generated by **Simulator**. We assume that both the set of states S and the set of actions A are finite subspaces of integer manifolds, i.e., $S \subset \mathbb{Z}^n$ and $A \subset \mathbb{Z}^m$, where n and m are the dimensions of the states and actions manifold, respectively. In the following, we call *state-space* the set of states S and *action-space* the set of actions A . Since both the state-space and the action-space are finite, a table Q of size $|S| \times |A|$ is sufficient to represent the *quality* $Q(s, a) \in \mathbb{R}$ of action a in state s , i.e., a real number quantifying how “desirable” is a in state s . Once learning is over, a policy can be extracted from the Q -table in a variety of ways. In our case studies, we consider stochastic policies obtained with a *softmax criterion* [15]

$$\pi_Q(s, a) = \frac{e^{\frac{Q(s, a)}{\beta}}}{\sum_{\alpha \in A} e^{\frac{Q(s, \alpha)}{\beta}}} \quad (2)$$

where β is called *temperature parameter*. For $\beta \rightarrow \infty$ (high temperature), all actions are equally probable, whereas for $\beta \rightarrow 0$ (low temperature), the probability of the action with the highest Q -value tends to 1. In the following, when no ambiguity arises, we will drop the subscript Q and leave the dependency of π on Q implicit.

The core algorithm of **Learner** is presented in Figure 1 (right). Here we can see that the algorithm, after initializing the Q -table by zeroing all entries, iterates over a series of episodes N_e fixed by the user. In each episode — inside the outer **for all** loop (lines 3-16) — and at each time step — inside the inner **repeat-until** loop — the j -th state s_j is received from the simulator (line 7), the next action a_j is computed using **COMPUTEPOLICY** (line 8), and it is sent to the simulator

(line 9). In particular, COMPUTEPOLICY selects an action a_j considering the Q -table and according to the softmax criterion (2). Once information about the next state s_{j+1} and related flags f_{j+1} is received (line 10-11), the computation of r_{j+1} is performed by the function COMPUTEREWARD (line 11), and the current policy is changed by UPDATE according to the experience E_j . In particular, for all $s \in S$ and $a \in A$, UPDATE estimates $Q(s, a)$ iteratively as

$$Q(s_j, a) \leftarrow Q(s_j, a) + \alpha [r_{j+1} + \gamma Q(s_{j+1}, a_{j+1}) - Q(s_j, a)] \quad (3)$$

where γ is the discount factor defined in (1) and α is a positive *step size parameter* which weights the temporal difference $[r_{j+1} + \gamma Q(s_{j+1}, a_{j+1}) - Q(s_j, a)]$ against the previous value of $Q(s_j, a)$. If the episode ends with s_{j+1} , then $Q(s_{j+1}, a_{j+1}) = 0$ for all actions $a \in A$. This definition of UPDATE makes LEARN an implementation of the *SARSA* [14] RL algorithm, a well known *temporal difference* method — see Chapter 3 of [15]. Given the definitions above, in order to specify the learning strategy in full, we need to define COMPUTEREWARD, and the contents of S and A which are specific to the actual learning problem.

The combination of the original MDP and a policy π computed by LEARN yields a *discrete-time Markov chain* (DTMC) describing all possible interactions of the agent with the environment. Following [9], given a set of atomic propositions AP , a DTMC is defined as a tuple $(W, \bar{w}, \mathbf{P}, L)$ where W is a finite set of *states*, and $\bar{w} \in W$ is the *initial state*; $\mathbf{P} : W \times W \rightarrow [0, 1]$ is the *transition probability matrix*; and $L : W \rightarrow 2^{AP}$ is the *labeling function*. An element $\mathbf{P}(w, w')$ gives the probability of making a transition from state w to state w' with the requirement that $\sum_{w' \in W} \mathbf{P}(w, w') = 1$ for all states $w \in W$. A terminating (absorbing) state w is modeled by letting $\mathbf{P}(w, w) = 1$. The labeling function maps every state to a set of propositions from AP which are true in that state. Safety assurance is expressed in terms of Probabilistic Computational Tree Logic (PCTL) formulas, whose syntax is defined considering the set Σ of *state formulas*, the set Π of *path formulas*. Σ is defined inductively as:

- if $\varphi \in AP$ then $\varphi \in \Sigma$;
- $\top \in \Sigma$; if $\alpha, \beta \in \Sigma$ then also $\alpha \wedge \beta \in \Sigma$ and $\neg \alpha \in \Sigma$
- $\mathcal{P}_{\bowtie p}[\psi] \in \Sigma$ where $\bowtie \in \{\leq, <, \geq, >\}$, $p \in [0, 1]$ and $\psi \in \Pi$.

Notice that path formulas only occur as the parameter of the *probabilistic path operator* $\mathcal{P}_{\bowtie p}[\psi]$. The set Π contains exactly the expressions of type $X\alpha$ (*next*), $\alpha\mathcal{U}^{\leq k}\beta$ (*bounded until*) and $\alpha\mathcal{U}\beta$ (*until*) where $\alpha, \beta \in \Sigma$ and $k \in \mathbb{N}$. We also abbreviate $\top\mathcal{U}\beta$ as $\mathcal{F}\beta$ (*eventually*) and $\top\mathcal{U}^{\leq k}\beta$ as $\mathcal{F}^{\leq k}\beta$ (*bounded eventually*). A *path* τ is defined as a non-empty sequence of states $w_0, w_1, w_2, \dots$ where $w_i \in W$ and $\mathbf{P}(w_i, w_{i+1}) > 0$ for all $i \geq 0$. We denote the i th state of τ by $\tau(i)$. Paths can be either finite or infinite. In case of finite paths, the *length* of the path is just the number of states in the path, and we denote with $Path_w$ the set of (infinite) paths starting from w .

Given a DTMC $\mathcal{M} = (W, \bar{w}, \mathbf{P}, L)$ and a state $w \in W$, let $\mathcal{M}, w \models \varphi$ denote that $\varphi \in \Sigma$ is satisfied in w . Similarly, we write $\mathcal{M}, \tau \models \psi$ for a path τ of $\mathcal{M}$ satisfying a formula $\psi \in \Pi$. When $\mathcal{M}$ is clear from the context, we write $w \models \varphi$

and $\tau \models \psi$, respectively. The semantics of path formulas can thus be defined as follows:

- $\tau \models X\alpha$ exactly when $\tau(1) \models \alpha$;
- $\tau \models \alpha \mathcal{U}^{\leq k} \beta$ exactly when there is some $i \leq k$ such that $\tau(i) \models \beta$ and $\tau(j) \models \alpha$ for all $j < i$;
- $\tau \models \alpha \mathcal{U} \beta$ exactly when there is some $k \geq 0$ such that $\tau \models \alpha \mathcal{U}^{\leq k} \beta$.

Accordingly, the semantics of state formulas can be stated as follows:

- $w \models a$ with $a \in AP$ iff $a \in L(w)$;
- $w \models \top$ for all $w \in W$; $w \models \alpha \wedge \beta$ exactly when $w \models \alpha$ and $w \models \beta$; $w \models \neg \alpha$ exactly when $w \models \alpha$ does not hold;
- $w \models \mathcal{P}_{\bowtie p}[\psi]$ exactly when $Prob_w(\{\tau \in Path_w \mid \tau \models \psi\}) \bowtie p$

Intuitively, $Prob_w(\dots)$ is the probability that the paths in $Path_w$ that also satisfy ψ are taken. $Prob_w(\dots)$ is provably measurable for all possible PCTL path formulas, and it can be computed by the transition probability matrix $\mathbf{P}$ — see [9] for details.

3 Verification and repair of learned policies

Let us assume that states and transitions explored during LEARN are logged in a table $T : S \times A \rightarrow 2^{S \times \mathbb{N}}$, where S is the state-space and A is the action-space. For each state $s \in S$ and action $a \in A$, we have that $T(s, a) = \{(s_1; n_1), \dots, (s_k; n_k)\}$ where $s_i \in S$ for all $1 \leq i \leq k$, and n_i is the number of times state s_i occurred as next state in a transition from state s on action a ; each cell (s, a) of T stores only the pairs $(s; n_i)$ such that $n_i > 0$, i.e., state s_i appeared at least once in a transition from an explored state s on action a . For all actions $a \in A$, if the state $s \in S$ was explored, but had no outgoing transitions, then $T(s, a) = \emptyset$; if a state was not explored, then $T(s, a) = \perp$.⁵ Depending on whether (un)safety is defined for states or transitions, we also must keep track of unsafe states/transitions that were visited/executed during learning. If unsafety is in terms of states, i.e., the state itself is undesirable, then we need a table $F : S \rightarrow \{\text{TRUE}, \text{FALSE}\}$, where $F(s) = \text{TRUE}$ exactly when state s is found to be unsafe during learning. On the other hand, if transitions can be unsafe, i.e., the action itself is undesirable, then we consider a table $F : S \times A \rightarrow \{\text{TRUE}, \text{FALSE}\}$ where, for each explored state $s \in S$ and action $a \in A$, we log whether the specific action a is found to be unsafe when started from state s .

3.1 Encoding behaviors and safety

We combine tables T and F with the Q -table returned by LEARN to obtain a DTMC $\mathcal{M} = (W, \bar{w}, \mathbf{P}, L)$. The set AP of atomic propositions has just one

⁵ Table T can handle (i) deterministic actions, i.e., a is such that $\forall s \in S. |T(s, a)| = 1$; (ii) nondeterministic actions, i.e., a is such that for all $s \in S$, both $|T(s, a)| = k$ and $n_i = 1/k$ for all $1 \leq i \leq k$; and (iii) probabilistic actions, i.e., arbitrary values of n_i .

element *bad* which denotes unwanted states. The construction of $\mathcal{M}$ is the following:

1. The set of states W is initialized to explored states logged in T , i.e., all the states $s \in S$ such that for all actions $a \in A$ we have $T(s, a) \neq \perp$, plus three additional states $w_0, w_{ok}, w_{bad} \notin S$. The initial state $\bar{w}$ is w_0 . For all $v, w \in W$, we initialize $\mathbf{P}(v, w) = 0$, and $L(w) = \{\}$.
2. Let $I \subseteq S$ be the set of *source states* in T , i.e., $s \in I$ exactly when $(s, n) \notin T(s', a)$ for all $s' \in S$, $n \in \mathbb{N}$ and $a \in A$. We make w_0 the unique initial state by requiring $\mathbf{P}(w_0, w) = \frac{1}{|I|}$ for all $w \in I$, meaning that w_0 has nondeterministic transition to every source state.
3. Given two states $v, w \in W$ such that $v, w \notin \{w_0, w_{ok}, w_{bad}\}$, the probability of transition is defined as

$$\mathbf{P}(v, w) = \sum_{a \in A} \pi_Q(v, a) \cdot p_T(v, a, w) \quad (4)$$

where π_Q is the stochastic policy extracted from Q , and $p_T(v, \hat{a}, w)$ is the probability of observing w as a successor of v given that action $\hat{a}$ was taken. Assuming $T(v, \hat{a}) = \{(w_1; n_1), \dots, (w_k; n_k)\}$ we have

$$p_T(v, \hat{a}, w) = \begin{cases} \frac{n_w}{\sum_{i=1}^k n_i} & \text{if } (w; n_w) = (w_j; n_j) \text{ for some } 1 \leq j \leq k \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

We define $K \subseteq S$ to be the set of *sink states* in T , i.e., $s \in K$ exactly when $T(s, a) = \emptyset$ for all $a \in A$. If the notion of unsafety is attached to transitions, then the construction of $\mathcal{M}$ is completed as follows:

4. The state w_{ok} is an abstraction of all sink states that can be reached by at least one safe transition, and $\mathbf{P}(w_{ok}, w_{ok}) = 1$. For each state v , let $safe(v) \subseteq K$ be the set of sink states that can be reached from v via a safe transition, i.e., all states $w \in K$ such that there exist $a \in A$ with $(w; n_w) \in T(v, a)$ and $F(v, a) = \text{FALSE}$; for all states v such that $safe(v) \neq \emptyset$, we set $\mathbf{P}(v, w_{ok}) = \sum_{w \in safe(v)} \mathbf{P}(v, w)$ and then $\mathbf{P}(v, w) = 0$ for all $w \in safe(v)$.
5. The state w_{bad} is an abstraction for all states – including non-sink ones – that can be reached by at least one unsafe transition. Therefore we set $\mathbf{P}(w_{bad}, w_{bad}) = 1$ and $L(w_{bad}) = \{bad\}$. For each state v , $unsafe(v) \subseteq K$ is the set of sink states that can be reached from v via an unsafe transition, i.e., all states $w \in K$ such that there exist $a \in A$ with $(w; n_w) \in T(v, a)$ and $F(v, a) = \text{TRUE}$; for all states v such that $unsafe(v) \neq \emptyset$, we set $\mathbf{P}(v, w_{bad}) = \sum_{w \in unsafe(v)} \mathbf{P}(v, w)$ and then $\mathbf{P}(v, w) = 0$ for all $w \in unsafe(v)$.

On the other hand, if the notion of unsafety is attached to states, $\mathcal{M}$ is completed following alternative steps:

4. The state w_{ok} is an abstraction of all sink states $s \in K$ such that $F(s) = \text{FALSE}$. We make w_{ok} absorbing by setting $\mathbf{P}(w_{ok}, w_{ok}) = 1$. For each state

v , let $safe(v) \subseteq K$ be the set of all safe sink states that can be reached from v , i.e., all states $w \in K$ such that there exist $a \in A$ with $(w; n_w) \in T(v, a)$ and $F(w) = \text{FALSE}$; for all states v such that $safe(v) \neq \emptyset$, we set $\mathbf{P}(v, w_{ok}) = \sum_{w \in safe(v)} \mathbf{P}(v, w)$ and then $\mathbf{P}(v, w) = 0$ for all $w \in safe(v)$.

5. Symmetrically to w_{ok} , the state w_{bad} is an abstraction for all unsafe sink states. Therefore we set $\mathbf{P}(w_{bad}, w_{bad}) = 1$ and $L(w_{bad}) = \{bad\}$. For each state v , $unsafe(v) \subseteq K$ is the set of all unsafe sink states that can be reached from v , i.e., all states $w \in K$ such that there exist $a \in A$ with $(w; n_w) \in T(v, a)$ and $F(w) = \text{TRUE}$; for all states v such that $unsafe(v) \neq \emptyset$, we set $\mathbf{P}(v, w_{bad}) = \sum_{w \in unsafe(v)} \mathbf{P}(v, w)$ and then $\mathbf{P}(v, w) = 0$ for all $w \in unsafe(v)$.

In both cases, we end up removing *orphan* states, i.e., states $w \in W$ with $w \neq w_0$ such that $\mathbf{P}(v, w) = 0$ for all $v \in W$ with $v \neq w$. For all such states w , we also remove outgoing connections by setting $\mathbf{P}(w, v) = 0$ for every $v \in W$. The process is repeated until no orphan state and related transitions are left.

Given the construction above, while the concrete meaning of safety will depend on the way F is computed, from an abstract point of view we must ensure that the probability of reaching w_{bad} is low, i.e., check

$$\mathcal{M}, w_0 \models \mathcal{P}_{<\sigma}[\mathcal{F} \text{ bad}] \quad (6)$$

where $\sigma \in [0, 1]$ is a safety assurance probability. Typical assertions that can be checked in this context include, e.g., the probability of a bad state is less than 1%, i.e., checking that $\mathcal{M}, w_0 \models \mathcal{P}_{<0.01}[\mathcal{F} \text{ bad}]$ holds. A control policy such that property (6) cannot be verified for small values of σ is clearly undesirable, also in the cases where low-level protection mechanism prevent actual failures to occur.

3.2 Fixing policies

We define *policy repair* as a verification-based procedure that modifies the Q -table to fix the corresponding policy. We assume that the DTMC $\mathcal{M}$ is obtained as described previously, and it is loop-free with the only exception of absorbing states w_{ok} and w_{bad} . Postulating the absence of loops may limit the applicability of repair in the general case. However, out of our three case studies, only learning in *grid world* has a non-negligible chance to yield optimal policies which induce cycles in the DTMC. In the other two cases, loop-free DTMCs are either the natural outcome, or they can be obtained without loss of generality or optimality. As for *grid world*, we observe that cycles in the DTMC are possible only because the straightforward encoding of *grid world* does not penalize useless energy dissipation, e.g., when “bouncing back” from the grid borders. Since this is quite unrealistic, we modify the learned model so that inefficient looping behavior is removed, and our method becomes applicable also in this case.

Repair is driven by counterexamples that witness the violation of property (6) for some threshold σ . Since there are no infinite paths in $\mathcal{M}$, property (6) is violated only if σ is less than the probability of choosing a path $\tau = w_0, w_1, \dots, w_k$

such that $w_k = w_{bad}$. The *probability mass* of a path τ is defined as

$$Prob(\tau) = \prod_{j=1}^k \mathbf{P}(w_{j-1}, w_j) \quad (7)$$

and extended to a set of paths $\mathcal{C} = \{\tau_1, \dots, \tau_n\} \subseteq Path_{w_0}$ as $Prob(\mathcal{C}) = \sum_{i=1}^n Prob(\tau_i)$. If, for all $1 \leq i \leq n$, we have $\tau_i = w_{i,0}, w_{i,1}, \dots, w_{i,k_i}$ with $w_{i,0} = w_0$, $w_{i,k_i} = w_{bad}$ and $Prob(\mathcal{C}) > \sigma$, then $\mathcal{C}$ is a counterexample of property (6). Notice that a counterexample $\mathcal{C}$ is unique only if it contains all paths starting with w_0 and ending with w_{bad} , and for all $\mathcal{C}' \subset \mathcal{C}$ we have that $Prob(\mathcal{C}') < \sigma$. Given our assumptions, the counterexample $\mathcal{C} = \{\tau_1, \dots, \tau_n\}$ is a DAG such that $|\mathcal{C}| \leq |Path_{w_0}|$. However, unless learning is highly ineffective, we expect that $|\mathcal{C}| \ll |Path_{w_0}|$, i.e., focusing on the counterexample is an effective way to limit the repair procedure to a handful of “critical” paths. However, focusing on the counterexample is just the first step in verification-based repair, and we also need to find an effective way to alter the policy, i.e., the Q -values, so that probability masses are shifted in the DTMC and property (6) is satisfied. Our implementation leverages two key ideas to accomplish this. The first one is that transitions between states which are “far” from w_{bad} should be discouraged less than those that are “near” w_{bad} . The second idea is that probabilities should “flow away” from paths ending with w_{bad} towards paths ending with w_{ok} . Altering transitions which are close to w_{bad} helps to keep changes to the policy as local as possible, and to avoid wasting useful learning results. Shifting probabilities towards paths ending with w_{ok} avoids moving probability mass from one unsafe path to another, yielding no global improvement.

We formalize the intuitions above with the following definitions. Given a path $\tau = w_0, \dots, w_k$ such that $\tau \in \mathcal{C}$, let $\tau^{-1}(w) : W \rightarrow \{0, \dots, k\}$ be the *index* of w in τ , i.e., $\tau^{-1}(w) = i$ if $w = w_i$ in τ ; The *successor of w along τ* , denoted by $succ_\tau(w)$, is the state v such that $\tau^{-1}(v) = \tau^{-1}(w) + 1$. Given a counterexample $\mathcal{C}$ and path $\tau = w_0, \dots, w_k$ such that $\tau \in \mathcal{C}$, we know that $L(w) = \{bad\}$ only if $w = w_k$. Therefore, we can define the *distance to the bad state along τ* as $b_\tau(w) = k - \tau^{-1}(w)$. Given a counterexample $\mathcal{C}$ and a path $\tau \in \mathcal{C}$, for every state $w \in \tau$ we consider a *penalty* function

$$z_\tau(w) = (1 - Prob(\tau)) \cdot (1 - e^{-\frac{b_\tau(w)}{\delta}}) \quad (8)$$

where $\delta \in \mathbb{R}^+$ is a real constant modulating the decay of the exponential for growing values of $b_\tau(w)$: all other aspects being equal, the higher is δ , the slower is the convergence of $z_\tau(w)$ to $1 - Prob(\tau)$ for increasing values of $b_\tau(w)$. Notice that, given two states w, w' along the same path τ such that $b_\tau(w) < b_\tau(w')$, we have $z_\tau(w) > z_\tau(w')$; given two paths $\tau, \tau' \in \mathcal{C}$ such that $Prob(\tau) > Prob(\tau')$, for all states w along τ and all states w' along τ' , if $b_\tau(w) = b_{\tau'}(w')$ we have that $z_\tau(w) > z_{\tau'}(w')$.

Given a model $\mathcal{M}$ built out of tables Q, T, F as described in Section 3, a counterexample $\mathcal{C} = \{\tau_1, \dots, \tau_n\}$ of the safety property (6) for a given threshold σ , and an iteration parameter $\theta \in \mathbb{N}$, repair is implemented by the following algorithm:

1. Make a copy Q' of the (current) Q -table Q .
2. Choose $\tau \in \mathcal{C}$ such that $Prob(\tau) \geq Prob(\tau')$ for all $\tau' \in \mathcal{C}$, i.e., τ is the path in $\mathcal{C}$ with the largest probability mass.
3. For each $w_i \in \tau$ ($0 \leq i < k$), compute $b_\tau(w_i)$ and let $v = succ_\tau(w)$; find the action $a \in A$ such that $(v; n_v) \in T(w, a)$ and modify $Q'(w, a)$ as follows

$$Q'(w, a) = \begin{cases} Q'(w, a) \cdot z_\tau(w) & \text{if } Q'(w, a) \geq 0 \\ Q'(w, a) \cdot \frac{1}{z_\tau(w)} & \text{if } Q'(w, a) < 0 \end{cases}$$

4. Let $\mathcal{C} = \mathcal{C} \setminus \{\tau\}$; go to step (2), unless $\mathcal{C} = \emptyset$ or more than θ paths were considered.
5. Let $\mathcal{M}'$ be the DTMC obtained by T , F and Q' in the usual way; check if $\mathcal{M}'$ fulfills (6); if so, then stop, otherwise a new counterexample $\mathcal{C}'$ is available: let $\mathcal{C} = \mathcal{C}'$, $\mathcal{M} = \mathcal{M}'$, $Q = Q'$ and go to step (1).

Two parameters control the repair algorithm, namely δ , which controls how strongly state-action values are affected depending on their proximity to bad states, and θ , i.e., the number of paths considered before checking again for safety. We expect that large values of δ may result in sharp improvements on safety (high effectiveness), whereas small values may result in less disruptive policy repairs (low invasiveness). As for θ , examining a small number of paths at a time, may require several repair cycles to “fix” the policy (low efficiency), whereas examining a large batch of paths, may alter more than what is necessary and possibly disrupt useful learning (high invasiveness). In all cases, a fixed value of θ guarantees the termination of the repair algorithm. Clearly, when it comes to effectiveness and invasiveness outcomes, both parameters must be tuned to trade-off between competing but desirable conditions. We consider this problem mainly from an empirical point of view, but the results presented in the next section suggest that such trade-offs can be achieved for a variety of cases.

4 Experimental analysis

Verification of policies is carried out with state-of-the-art probabilistic model checkers, namely COMICS [1] (version 1.0), MRMC [7] (version 1.4.1), and PRISM [8] (version 4.0.3). Concerning the automated repair procedure, we report the results related to COMICS only, because MRMC does not implement counterexample generation, and in PRISM this is still a beta-stage feature. All tools are run in their default configuration with the exception of COMICS, for which the option `--concrete` is selected instead of the default `--abstract`. Finally, verification and repair are performed on a Intel Core i3-2330M quad core at 2.20 GHz with 4GB RAM, equipped with Ubuntu 12.04 LTS 64 bit.

4.1 Air hockey

Air hockey is played by two players. They use round paddles (mallets) to hit a flat round puck across a low-friction table. At each end of the table there is

a goal area. The objective of the game is to hit the puck so that it goes into the opponent’s goal (offense play), and to prevent the opponent to score into your own goal (defense play). At the beginning of each episode, the puck is fired from a random point in vicinity of the opponent’s goal area, with speed and inclination chosen uniformly at random within some practically feasible range. An episode ends in one of the following ways: (i) the puck hits the border on the opponent’s side of the table, (ii) a time-out occurs, or (iii) a goal is scored.

With reference to a Cartesian system centered in the robot’s goal area, the position of the puck is represented by the couple (p, α) , where p is the perpendicular dropped from the origin on the trajectory line of the puck, and α is the angle made by this perpendicular with respect to the abscissa. A state is a vector (p, α, θ) , where θ is the current robot’s mallet angle with respect to a polar system also centered in the robot’s goal area.⁶ The parameters p and α are quantized over 58 and 37 values, respectively, and θ is quantized over 31 values, yielding 66526 possible states. The flag vector f contains the reset signal plus three additional elements: $f(\text{mallet})$ is raised when the robot’s mallet hits the puck; $f(\text{goal})$ signals a goal scored by the opponent; and $f(\text{speed})$ is raised when robot’s velocity exceeds a critical threshold. **Simulator** receives from **Learner** the action as $(\Delta\theta, \Delta t)$ where $\Delta\theta$ is the angular displacement suggested for the robot mallet, and Δt is the time to achieve it. We keep Δt fixed to 0.02 seconds for the sake of simplicity, and $\Delta\theta$ is quantized over 31 different values. To complete the specification of **LEARN** in this case, we define the auxiliary function **COMPUTE-REWARD** so that it awards a baseline value $R \in \mathbb{R}^+$ to the agent when it saves a goal. A smaller reward of $\frac{R}{5}$ is given when the puck is hit, even if the current trajectory is not leading to a goal. A negative reward (penalty) $-R$ is assigned when the puck hits the robot’s goal area. Furthermore, a penalty $-\frac{R}{5}$ is assigned when $f(\text{speed})=\text{TRUE}$. Finally, to speed up learning, a small positive reward $\frac{R}{50}$ is assigned to each non-terminal state.

The learning stage is composed of 10000 episodes, and we report that learning converges to a reasonable performance after about 5000 episodes. The percentage of saved goals vs. potential goals is always greater than 0.8 after 2000 episodes, and this lower bound raises to 0.95 after 5000 episodes. As for safety, after 5000 episodes, the percentage of unsafe transitions is always less than 0.02, and this upper bound does not decrease in a substantial way even after 10000 learning episodes.

The obtained DTMC encoding has 4313 states and 8850 transitions. All the model checkers that we consider are able to compute the safety threshold σ , agreeing on a value $\sigma = 0.0243$. In terms of CPU time, their performance is quite different, in that **COMICS** and **MRMC** complete in less than 0.01s, whereas **PRISM** requires 3.18s to check the property — with additional 164.14s to build the model. In this regard, it is important to notice that **COMICS** and **MRMC** require an explicitly built DTMC as input, whereas **PRISM** allows for an implicit

⁶ The radius ρ is not needed because we keep it fixed throughout learning and simulation. In a pure defense play, this choice does not hamper the robot’s ability to defend the goal area.

$\sigma \backslash \delta$	1			5			10		
	τ	Eff.	Time	τ	Eff.	Time	τ	Eff.	Time
0.01	17	0.967	12.93	7	0.968	5.26	7	0.968	5.57
0.001	48	0.967	37.22	25	0.970	18.95	25	0.961	19.09
0.0001	75	0.968	59.84	39	0.966	29.89	39	0.964	29.35

Table 1: Automated repair. Columns and rows are labeled with values of δ — the decay coefficient in Eq. 8 — and σ — the target safety threshold. Each group is composed of three columns. In the first one (“ τ ”), we report the total amount of counterexamples involved in the repair. In the second column (“Eff.”), we report the effectiveness of the learning agent, i.e., the mean value over 10 random tests of the ratio of saved goals versus potential goals after repairing. Finally, in the last column (“Time”) we report the CPU time (in seconds) spent by the repair procedure.

declaration which is more user friendly and compact, but requires extra time to be compiled into a DTMC.

In Table 1, we report the empirical results related to the automated repair procedure. We checked three different safety thresholds, namely $\sigma = \{0.01, 0.001, 0.0001\}$ against three different values of δ , namely 1, 5, and 10. In this experiment, $\theta = 1$, i.e., we re-check safety after taking into account every path of $\mathcal{C}$. As mentioned before, our starting point is $\sigma = 0.0243$, so every value of σ in Table 1 requires fixing the agent. The initial performance score of the agent is 0.967, i.e., nearly 97% of the potential goals are saved. Looking at Table 1, we can see that robot’s performance does not change in a noticeable way, and thus the repair procedure is not invasive in these configurations. Furthermore, repair guarantees a safety level which is two orders of magnitude smaller than the starting one and, in this sense, it is also highly effective.

4.2 iCub reaching

Our second case study is about the humanoid iCub [12] learning to reach for an object (*target*) placed on a table, without damaging other objects (*obstacles*) in the process. At the beginning of each episode, iCub starts moving towards the target trying to execute high-level discrete actions. A low-level controller ensures that the resulting trajectory of the robot is smooth and efficient. An episode ends either because (i) the target is reached, or (ii) an obstacle is hit. In all these cases, the flag $f(reset)$ becomes TRUE. In the simulator, the current state is a vector $(x, y, z, o) \in S$, where the component (x, y, z) represents the Cartesian coordinates of iCub’s hand in a system having origin in its torso, and o represents the orientation of the hand. In the current implementation, the coordinates have the following ranges (in meters): $x \in [-0.35, -0.1]$, $y \in [-0.15, 0.25]$, $z \in [-0.05, 0.15]$ with discretization step of 0.01m, 0.02m and 0.04m, respectively. The orientation $o \in \{-1, 0, 1\}$ considers only three positions:

“face down”, “face up”, and “face left”, respectively. This yields a total of 9828 possible states. The flag vector contains also two additional elements: the signal $f(\text{collision})$ which reveals a collision with the obstacle, and the signal $f(\text{posture})$ which signals whenever reaching did not happen with $o = 1$. **Simulator** receives the action $a \in A$ as $(\Delta x, \Delta y, \Delta z, \Delta o, t)$ where the first four components are the change in current Cartesian coordinates and orientation, respectively. The fifth component is the time to execute the action. Given the quantization of the values, there are a total of 405 possible actions. Contrast this with **air hockey** — where the number of states is about 7 times larger, and the number of actions is about 10 times smaller — to see that **air hockey** is somewhat complementary to it in its abstract formulation. To describe **COMPUTERREWARD**, we first define the *minimum distance* $d(s, t)$ between any two states $s, t \in S$ recursively as

$$d = \begin{cases} 0 & \text{if } s = t \\ 1 + \min_{a \in A_s} d(\text{succ}(s, a), t) & \text{otherwise.} \end{cases} \quad (9)$$

where A_s is the set of actions which is enabled from state s , and $\text{succ}(s, a)$ is the state which succeeds s after executing action a . The value R returned by **COMPUTERREWARD** is defined as follows

$$R = \begin{cases} R_t + f(T) & \text{if only } f(\text{reset})=\text{TRUE} \\ R_o & \text{if } f(\text{collision})=\text{TRUE} \\ R_n + g(d) & \text{if all flags are FALSE} \end{cases} \quad (10)$$

where T is the number of time-steps required to reach the goal, $f : \mathbb{N} \rightarrow \mathbb{R}$ is a monotonic bounded-downwards function defined as $f(T) = \max(-C_1, 1 - e^{C_2 \cdot T})$, d is the distance of the current state from the goal, and $g : \mathbb{N} \rightarrow \mathbb{R}$ is a monotonic bounded-downwards function defined as $g(d) = \min(K_1, \frac{K_2}{1 - e^{-d}})$ with $K_1 > K_2 > 0$. The constants $C_1 > 0$ and C_2 that can be seen as negative utility for longer actions, and the constant K_2 represents how preferable is a closer location with respect to a farther one considering the goal. R_n is a small penalty imposed on each action in order to quicken convergence of learning, R_t is the target-reaching reward, and R_o is the negative reward for hitting the obstacle — in our current implementation $R_t = 25$ ($C_1 = 1$ and $C_2 = 2$), $R_o = -2$, $R_n = -0.1$ ($K_1 = 1$ and $K_2 = 0.05$).

After a learning stage composed of about 4500 episodes, the robot can reach the ball while simultaneously avoiding contact with the obstacle in most cases. The related DTMC has 590 states and 2148 transitions. Also in this case, the model checkers are able to compute a safety threshold σ , agreeing on a value $\sigma = 0.0081$. **COMICS** and **MRMC** complete the check in less than 0.01 CPU seconds, whereas **PRISM** takes about 27 CPU seconds to complete. The motivation of this difference are the same as in the **air hockey** case study. Concerning repair, we report that **COMICS** spends a mean value of 0.16 CPU seconds considering 10 random starting positions of the iCub arm.

4.3 Grid world

Our third case study is the “classical” simplified navigation problem for a mobile robot, where the environment is represented by a grid of $n \times n$ cells, and the

		$n = 5$	$n = 10$	$n = 25$	$n = 50$	$n = 75$	$n = 100$
# states		22	89	559	2439	5550	9898
# transitions		51	413	2701	11787	27126	48768
σ		0.796	0.144	0.005	0.013	0.008	0.011
TTV	COMICS	0.001	0.001	0.020	0.250	1.36	6.260
	MRMC	0.001	0.001	0.002	0.011	0.042	0.097
	PRISM	0.025	0.214	5.934	161.971	985.291	3243.835
TTR	COMICS	0.683	0.563	1.032	4.921	14.492	49.004

Table 2: Verification and repair in the grid world. For each size n , we report the number of states and transitions in the DTMC and the initial safety threshold σ . “TTV” is the time to compute σ and “TTR” is the time to repair the policy in order to have $\sigma < 10^{-5}$ for all values of n .

robot can move one cell at a time, taking care to avoid pits, i.e., forbidden cells. We assume that start and target positions are fixed in the upper-left corner and bottom-right corner, respectively, and they never contain a pit. Also, we postulate that there exist at least one path from the start to the target cell. Learning in the grid world is conceptually simpler than the other two case studies, but the domain is easily scalable which enables us to control the complexity of learning, verification and repair tasks. The episode begins with the robot in the start position and it ends either because (i) the target is reached, or (ii) a pit is found. In both cases $f(reset)=\text{TRUE}$, and in case of failure then also $f(pit)=\text{TRUE}$. Finally, the robot bounces back when hits against a wall, i.e., the cells related to the grid borders. In the simulator, the current state $s \in S$ is represented by a couple (x, y) of cell coordinates, which means that there are always n^2 states. An action a is simply one of eight possible moves in adjacent cells. The function `COMPUTEREWARD` returns -40 if the robot falls in a pit, 100 if the robot reaches the target, -20 in the case of reset, and -0.1 at each step. To assess scalability of our approach, we consider the above problem for $n \in \{5, 10, 25, 50, 75, 100\}$, and with n randomly distributed pits.

The result of the experiments for various sizes of the grid world are reported in Table 2. In particular, out of several samples considering various obstacle placements, we always choose the one yielding the highest σ , i.e., the most unsafe control policy. The time to learn a control policy ranges from minutes of CPU time for $n = 10$ to hours for $n = 100$. As it is reasonable to expect, the time to verify the DTMC grows with n for all the tools we consider. However, performance is not homogeneous: TTV of COMICS increases of one order of magnitude from $n = 25$ to $n = 50$, while TTV of PRISM increases of two orders of magnitude, and it is also consistently higher. Concerning MRMC, we can see that TTV increases much slower than both PRISM and COMICS. Our current conjecture is that MRMC data structures are more efficient than COMICS and PRISM ones on problems of this kind. If we consider repair, then we see that TTR is reasonable even on the largest domain – less than one minute of CPU time, including both counterexample

generation by COMICS, and the execution of the repair algorithms itself. Similarly to air hockey and icub reaching, the effectiveness of control policies is not hurt by repair. From these results, we can conclude that repair confirms a non invasive behaviour coupled with reasonable performance scalability.

References

1. E. Abrahám, N. Jansen, R. Wimmer, J. Katoen, and B. Becker. Dtmc model checking by scc reduction. In *Quantitative Evaluation of Systems (QEST), 2010 Seventh International Conference on the*, pages 37–46. IEEE, 2010.
2. A. Aziz, V. Singhal, F. Balarin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In *Computer Aided Verification*, pages 155–165. Springer, 1995.
3. J. Bagnell and S. Schaal. Special issue on Machine Learning in Robotics (Editorial). *The International Journal of Robotics Research*, 27(2):155–156, 2008.
4. C. Baier and M. Kwiatkowska. Preface to the special issue on probabilistic model checking. *Formal Methods in System Design*, 43(2):121–123, 2013.
5. Cicala G. and Khalili A. and Metta G. and Natale L. and Pathak S. and Pulina L. and Tacchella A. Engineering approaches and methods to verify software in autonomous systems. In *13th International Conference on Intelligent Autonomous Systems*, Advances In Intelligent Systems and Computing. Springer, 2014.
6. J. Gillula and C. Tomlin. Guaranteed Safe Online Learning via Reachability: tracking a ground target using a quadrotor. In *ICRA*, pages 2723–2730, 2012.
7. J. Katoen, I. Zapreev, E. Hahn, H. Hermanns, and D. Jansen. The ins and outs of the probabilistic model checker mrmc. *Performance evaluation*, 68(2):90–104, 2011.
8. M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. *Computer Performance Evaluation: Modelling Techniques and Tools*, pages 113–140, 2002.
9. M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. *Formal methods for performance evaluation*, pages 220–270, 2007.
10. M. Kwiatkowska, G. Norman, and J. Sproston. *Probabilistic model checking of the IEEE 802.11 wireless local area network protocol*. Springer, 2002.
11. M. Lahijanian, J. Wasniewski, S. B. Andersson, and C. Belta. Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3227–3232. IEEE, 2010.
12. G. Metta, L. Natale, F. Nori, G. Sandini, D. Vernon, L. Fadiga, C. von Hofsten, K. Rosander, M. Lopes, J. Santos-Victor, et al. The iCub Humanoid Robot: An Open-Systems Platform for Research in Cognitive Development. *Neural networks: the official journal of the International Neural Network Society*, 2010.
13. S. Pathak, L. Pulina, G. Metta, and A. Tacchella. Ensuring safety of policies learned by reinforcement: Reaching objects in the presence of obstacles with the icub. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 170–175. IEEE, 2013.
14. G. Rummery and M. Niranjan. *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.
15. R. Sutton and A. Barto. *Reinforcement Learning – An Introduction*. MIT Press, 1998.
16. C. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.