Run-Time Optimization for Learned Controllers Through Quantitative Games

Guy Avni[1(✉)], Roderick Bloem[2], Krishnendu Chatterjee[1], Thomas A. Henzinger[1], Bettina Könighofer[2], and Stefan Pranger[2]

[1] IST Austria, Klosterneuburg, Austria
guy.avni@ist.ac.at
[2] TU Graz, Graz, Austria

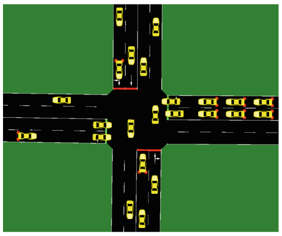
Abstract. A controller is a device that interacts with a plant. At each time point, it reads the plant's state and issues commands with the goal that the plant operates optimally. Constructing optimal controllers is a fundamental and challenging problem. Machine learning techniques have recently been successfully applied to train controllers, yet they have limitations. Learned controllers are monolithic and hard to reason about. In particular, it is difficult to add features without retraining, to guarantee any level of performance, and to achieve acceptable performance when encountering untrained scenarios. These limitations can be addressed by deploying quantitative run-time *shields* that serve as a proxy for the controller. At each time point, the shield reads the command issued by the controller and may choose to alter it before passing it on to the plant. We show how optimal shields that interfere as little as possible while guaranteeing a desired level of controller performance, can be generated systematically and automatically using reactive synthesis. First, we abstract the plant by building a stochastic model. Second, we consider the learned controller to be a black box. Third, we measure *controller performance* and *shield interference* by two quantitative run-time measures that are formally defined using weighted automata. Then, the problem of constructing a shield that guarantees maximal performance with minimal interference is the problem of finding an optimal strategy in a stochastic 2-player game "controller versus shield" played on the abstract state space of the plant with a quantitative objective obtained from combining the performance and interference measures. We illustrate the effectiveness of our approach by automatically constructing lightweight shields for learned traffic-light controllers in various road networks. The shields we generate avoid liveness bugs, improve controller performance in untrained and changing traffic situations, and add features to learned controllers, such as giving priority to emergency vehicles.

1 Introduction

The *controller synthesis* problem is a fundamental problem that is widely studied by different communities [42,44]. A controller is a device that interacts with a *plant*. In each point in time it reads the plant's state, e.g., given by sensor reading, and issues

This research was supported in part by the Austrian Science Fund (FWF) under grants S114 (RiSE/SHiNE), Z211-N23 (Wittgenstein Award), and M 2369-N33 (Meitner fellowship).

a command based on the state. The controller should guarantee that the plant operates correctly or optimally with respect to some given specification. As a running example, we consider a traffic light controller for a road intersection (see Fig. 1). The state of the plant refers to the state of the roads leading to the junction; namely, the positions of the cars, their speeds, their sizes, etc. A controller command consists of a light configuration for the junction in the next time frame. Specifications can either be qualitative, e.g., "it should never be the case that a road with an empty queue gets a green light", or quantitative, e.g., "the cost of a controller is the average waiting times of the cars in the junction".



$$\begin{tikzcd}
 & & (3,5,2,1) \\
(3,5,3,1) \arrow[r, "NS"] & (2,5,2,1) \arrow[ur, bend left, "\frac{1}{16}{,}\ (1,0,0,0)"] \arrow[r, "\frac{1}{16}{,}\ (0,1,0,0)"] \arrow[dr, bend right, "\frac{1}{16}{,}\ (0,0,1,0)"] & (2,5,2,1) \\
 & & (2,5,3,1) \\
 & & \vdots
\end{tikzcd}$$

Fig. 1. On the left, a concrete state depicted in the traffic simulator SUMO. On the right, we depict the corresponding abstract state with queues cut off at $k = 5$, and some outgoing transitions. Upon issuing action North-South, a car is evicted from each of the North-South queues. Then, we choose uniformly at random, out of the 16 possible options, the incoming cars to the queues, update the state, and cutoff the queues at k (e.g., when a car enters from East, the queue stays 5).

A challenge in controller synthesis is that, since the number of possible plant readings is huge, it is computationally demanding to find an optimal command, given a plant state. Machine learning is a prominent approach to make decisions based on large amounts of collected data [28,37]. It is widely successful in practice and takes an integral part in the design process of various systems. Machine learning has been successfully applied to train controllers [15,33,34] and specifically controllers for traffic control [20,35,39].

A shortcoming of machine-learning techniques is that the controllers that are produced are black-box devices that are hard to reason about and modify without a complete re-training. It is thus challenging, for example, to obtain worst-case guarantees about the controller, which is particularly important in safety-critical settings. Attempts to address this problem come from both the formal methods community [46], where verification of learned systems is extensively studied [24,29], and the machine-learning community, where guarantees are added during the training process using reward engineering [13,18] or by modifying the exploration process [11,19,38]. Both approaches require expertise in the respective field and suffer from limitations such as scalability for the first, and intricacy and robustness issues, for the second. Moreover, both techniques were mostly studied for safety properties.

Another shortcoming of machine-learning techniques is that they require expertise and a fine-tuning of parameters. It is difficult, for example, to train controllers that are robust to plant behaviors, e.g., a controller that has been trained on uniform traffic congestion meeting rush-hour traffic, which can be significantly different and can cause poor performance. Also, it is challenging to add features to a controller without retraining, which is both costly and time consuming. These can include permanent features, e.g., priority to public transport, or temporary changes, e.g., changes due to an accident or construction. Again, since the training process is intricate, adding features during training can have unexpected effects.

In this work, we use quantitative *shields* to deal with the limitations of learned or any other black-box controllers. A shield serves as a proxy between the controller and the plant. In each point in time, as before, the controller reads the state of the plant and issues a command. Rather than directly feeding the command to the plant, the shield first reads it along with an abstract plant state. The shield can then choose to keep the controller's command or alter it, before issuing the command to the plant. The concept of shields was first introduced in [30], where shields for safety properties were considered and with a qualitative notion of interference: a shield is only allowed to interfere when a controller error occurs, which is only well-defined when considering safety properties. We elaborate on other shield-like approaches in the Sect. 1.1.

Our goal is to automatically synthesize shields that optimize quantitative measures for black-box controllers. We are interested in synthesizing lightweight shields. We assume that the controller performs well on average, but has no worst-case guarantees. When combining the shield and the controller, intuitively, the controller should be active for the majority of the time and the shield intervenes only when it is required. We formalize the plant behavior as well as the interference cost using quantitative measures. Unlike safety objectives, where it is clear when a shield must interfere, with quantitative objectives, a non-interference typically does not have a devastating effect. It is thus challenging to decide, at each time point, whether the shield should interfere or not; the shield needs to balance the cost of interfering with the decrease in performance of not interfering. Automatic synthesis of shields is thus natural in this setting.

We elaborate on the two quantitative measures we define. The interaction between the plant, controller, and shield gives rise to an infinite sequence over $C \times \Gamma \times \Gamma$, where C is a set of plant states and Γ is a set of allowed actions. A triple $\langle c, \gamma_1, \gamma_2 \rangle$ means that the plant is in state c, the controller issues command γ_1, and the shield (possibly) alters it to γ_2. We use *weighted automata* to assign costs to infinite traces, which have proven to be a convenient, flexible, and robust quantitative specification language [14]. Our *behavioral score* measures the performance of the plant and it is formally given by a weighted automaton that assigns scores to traces over $C \times \Gamma$. Boolean properties are a special case, which include *safety* properties, e.g., "an emergency vehicle should always get a green light", and *liveness*, e.g., "a car waiting in a queue eventually gets the green light". An example of a quantitative score is the long-run average of the waiting times of the vehicles in the city. A second score measures the *interference* of a shield with a controller. It is given by a weighted automaton over the alphabet $\Gamma \times \Gamma$. A simple example of an interference score charges the shield 1 for every change of action and charges 0 when no change is made. Then, the score of an infinite trace can be phrased as the ratio of the time that the shield interferes. Using weighted automata we can specify more involved scores such as different charges for different types of alterations or even charges that depend on the past, e.g., altering the controller's command twice in a row is not allowed.

Given a probabilistic plant model and a formal specification of behavioral and interference scores, the problem of synthesizing an optimal shield is well-defined and can be solved by game theory. While the game-based techniques we use are those of discrete-event controller synthesis [3] in a stochastic setting with quantitative objectives, our set-up is quite different. In traditional controller synthesis, there are two entities; the controller and the adversarial plant. The goal is to synthesize a controller offline. In our setting, there are three entities: the plant, whose behavior we model probabilistically, the controller, which we treat as a black-box and model as an adversary, and the shield, which we synthesize. Note that the shield's synthesis procedure is done offline but it makes online decisions when it operates together with the controller and plant. Our plant model is formally given by a *Markov decision process* which is a standard model with which one models lack of knowledge about the plant using probability (see Fig. 1 and details in Example 1). The game is played on the MDP by two players; a shield and a controller, where the quantitative objective is given by the two scores. An optimal shield is then extracted from an optimal strategy for the shield player. The game we construct admits memoryless optimal strategies, thus the size of the shield is proportional to the size of the abstraction of the plant. In addition, it is implemented as a look-up table for actions in every state. Thus, the runtime overhead is a table look-up and hence negligible.

We experiment with our framework by constructing shields for traffic lights in a network of roads. Our experimental results illustrate the usefulness of the framework. We construct shields that consistently improve the performance of controllers, especially when exhibiting behavior that they are not trained on, but, more surprising, also while exhibiting trained behavior. We show that the use of a shield reduces variability in performance among various controllers, thus when using a shield, the choice of the parameters used in the training phase becomes less acute. We show how a shield can be used to add the functionality of prioritizing public transport as well as local fairness to a controller, both without re-training the controller. In addition, we illustrate how shields can add worst-case guarantees on liveness without a costly verification of the controller.

1.1 Related Work

A shield-like approach to adding safety to systems is called *runtime assurance* [47], and has applications, for example, in control of robotics [41] and drones [12]. In this framework, a switching mechanism alternates between running a high-performance system and a provably safe one. These works differ from ours since they consider safety specifications. As mentioned earlier, a challenge with quantitative specifications is that, unlike safety specifications, a non-interference typically does not have a devastating effect, thus it is not trivial to decide when and to what extent to interfere.

Another line of work is *runtime enforcement*, where an enforcer monitors a program that outputs events and can either terminate the program once it detects an error [45], or alter the event in order to guarantee, for example, safety [21], richer qualitative objectives [16], or privacy [26,49]. The similarities between an enforcer and a shield is in their ability to alter events. The settings are quite different, however, since the enforced program is not reactive whereas we consider a plant that receives commands.

Recently, formal approaches were proposed in order to restrict the exploration of the learning agent such that a set of logically constraints are always satisfied. This method can support other properties beyond safety, e.g., probabilistic computation tree logic (PCTL) [25,36], linear temporal logic (LTL) [1], or differential dynamic logic [17]. To the best of our knowledge, quantitative specifications have not yet been considered. Unlike these approaches, we consider the learned controller as a black box, thus our approach is particularly suitable for machine learning non-experts.

While MDPs and partially-observable MDPs have been widely studied in the literature w.r.t. to quantitative objectives [27,43], our framework requires the interaction of two players (the shield and the black-box controller) and we use game-theoretic framework with quantitative objectives for our solution.

2 Definitions and Problem Statement

2.1 Plants, Controllers, and Shields

The interaction with a *plant* over a concrete set of states C is carried out using two functionalities: PLANT.GETSTATE returns the plant's current state and PLANT.ISSUECOMMAND issues an action from a set Γ. Once an action is issued, the plant updates its state according to some unknown transition function. At each point in time, the *controller* reads the state of the plant and issues a command. Thus, it is a function from a history in $(C \times \Gamma)^* \cdot C$ to Γ.

Informally, a *shield* serves as a proxy between the controller and the plant. In each time point, it reads the controller's issued action and can choose an alternative action to issue to the plant. We are interested in light-weight shields that add little or no overhead to the controller, thus the shield must be defined w.r.t. an abstraction of the plant, which we define formally below.

Abstraction. An abstraction is a *Markov decision process* (MDP, for short) is $\mathcal{A} = \langle \Gamma, A, a_0, \delta \rangle$, where Γ is a set of actions, A is a set of abstract plant states, $a_0 \in A$ is an initial state, and $\delta : A \times \Gamma \to [0,1]^A$ is a probabilistic transition function, i.e., for every $a \in A$ and $\gamma \in \Gamma$, we have $\sum_{a' \in A} \delta(a, \gamma)(a') = 1$. The probabilities in the abstraction model our lack of knowledge of the plant, and we assume that they reflect the behavior exhibited by the plant. A *policy* f is a function from a finite history of states in A^* to the next action in Γ, thus it gives rise to a probabilistic distribution $\mathcal{D}(f)$ over infinite sequences over A.

Example 1. Consider a plant that represents a junction with four incoming directions (see Fig. 1). We describe an abstraction $\mathcal{A}$ for the junction that specifies how many cars are waiting in each queue, where we cut off the count at a parameter $k \in \mathbb{N}$. Formally, an abstract state is a vector in $\{0, \ldots, k\}^4$, where the indices respectively represent the North, East, South, and West queues. The larger k is, the closer the abstraction is to the concrete plant. The set of possible actions represent the possible light directions in the junction $\{\mathrm{NS}, \mathrm{EW}\}$. The abstract transitions estimate the plant behavior, and we describe them in two steps. Consider an abstract state $a = (a_1, a_2, a_3, a_4)$ and suppose the issued action is NS, where the case of EW is similar. We allow a car to cross the junction from each of the North and South queues and decrease the two queues. Let $a' = (\max\{0, a_1 - 1\}, a_2, \max\{0, a_3 - 1\}, a_4)$. Next, we probabilistically model incoming cars to the queues as follows. Consider a vector $\langle i_1, i_2, i_3, i_4 \rangle \in \{0,1\}^4$ that represents incoming cars to the queues. Let a'' be such that, for $1 \leq j \leq 4$, we add i_j to the j-th queue and trim at k, thus $a''_j = \min\{a'_j + i_j, k\}$. Then, in $\mathcal{A}$, when performing action NS in a, we move to a'' with the uniform probability $1/16$. □

We define shields formally. Let Γ be a set of commands, M a set of memory states, C and A be a set of concrete and abstract states, respectively, and let $\alpha : C \to A$ be a mapping between the two. A shield is a function SHIELD : $A \times M \times \Gamma \to \Gamma \times M$ together with an initial memory state $m_0 \in M$. We use PLANT to refer to the plant, which, recall, has two functionalities: reading the current state and issuing a command from Γ. Let CONT be a controller, which has a single functionality: given a history of plant states, the controller issues the command to issue to the plant. The interaction of the components is captured in the following pseudo code:

```
$m \leftarrow m_0 \in M$ and $\pi \leftarrow$ empty sequence.
**while** true **do**
    $c \leftarrow$ PLANT.GETSTATE() $\in C$
    $\gamma \leftarrow$ CONT.GETCOMMAND($\pi \cdot c$)
    $a = \alpha(c) \in A$        // generate abstract state for shield
    $\gamma', m' \leftarrow$ SHIELD($a, \gamma, m$)
    PLANT.ISSUECOMMAND($\gamma'$)
    $m \leftarrow m'$              // update shield memory state
    $\pi \leftarrow \pi \cdot \langle c, \gamma' \rangle$        // update plant history
**end while**
```

2.2 Quantitative Objectives for Shields

We are interested in two types of performance measures for shields. The *behavioral measure* quantifies the quality of the plant's behavior when operated with a controller and shield. The *interference measure* quantifies the degree to which a shield interferes with the controller. Formally, we need to specify values for infinite sequences, and we use *weighted automata*, which are a convenient model to express such values.

Weighted Automata. A weighted automaton is a function from infinite strings to values. Technically, a weighted automaton is similar to a standard automaton only that the transitions are labeled, in addition to letters, with numbers (weights). Unlike standard automata in which a run is either accepting or rejecting, a run in a weighted automaton has a value. We focus on limit-average automata in which the value is the limit average of the running sum of weights that it traverses. Formally, a weighted automaton is $\mathcal{W} = \langle \Sigma, Q, q_0, \Delta, cost \rangle$, where Σ is a finite alphabet, Q is a finite set of states, $\Delta \subseteq (Q \times \Sigma \times Q)$ is a deterministic transition relation, i.e., for every $q \in Q$ and $\sigma \in \Sigma$, there is at most one $q' \in Q$ with $\Delta(q, \sigma, q')$, and $cost : \Delta \to \mathbb{Q}$ specifies costs for transitions. A *run* of $\mathcal{W}$ on an infinite word $\sigma = \sigma_1, \sigma_2, \ldots$ is $r = r_0, r_1, \ldots \in Q^\omega$

such that $r_0 = q_0$ and, for $i \geq 1$, we have $\Delta(r_{i-1}, \sigma_i, r_i)$. Note that $\mathcal{W}$ is deterministic so there is at most one run on every word. The value that $\mathcal{W}$ assigns to σ is $\liminf_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} cost(r_{i-1}, \sigma_i, r_i)$.

Behavioral Score. A *behavioral* score measures the quality of the behavior that the plant exhibits. It is given by a weighed automaton over the alphabet $A \times \Gamma$, thus it assigns real values to infinite sequences over $A \times \Gamma$. In our experiments, we use a *concrete* behavioral score, which assigns values to infinite sequences over $C \times \Gamma$. We compare the performance of the plant with various controllers and shields w.r.t. the concrete score rather than the abstract score. With a weighted automaton we can express costs that change over time: for example, we can penalize traffic lights that change frequently.

Interference Score. The second score we consider measures the interference of the shield with the controller. An *interference* score is given by a weighted automaton over the alphabet $\Gamma \times \Gamma$. With a weighted automaton we can express costs that change over time: for example, interfering once costs 1 and any successive interference costs 2, thus we reward the shield for short interferences.

From Shields and Controllers to Policies. Consider an abstraction MDP $\mathcal{A}$. To ensure worst-case guarantees, we treat the controller as an adversary for the shield. Let $\textsc{Shield}$ be a shield with memory set M and initial memory state m_0. Intuitively, we find a policy in $\mathcal{A}$ that represents the interaction of $\textsc{Shield}$ with a controller that maximizes the cost incurred. Formally, an *abstract controller* is a function $\chi : A^* \to \Gamma$. The interaction between $\textsc{Shield}$ and χ gives rise to a policy $pol(\textsc{Shield}, \chi)$ in $\mathcal{A}$, which, recall, is a function from A^* to Γ. We define $pol(\textsc{Shield}, \chi)$ inductively as follows. Consider a history $\pi \in A^*$ that ends in $a \in A$, and suppose the current memory state of $\textsc{Shield}$ is $m \in M$. Let $\gamma = \chi(\pi)$ and let $\langle \gamma', m' \rangle = \textsc{Shield}(\gamma, a, m)$. Then, the action that the policy $pol(\textsc{Shield}, \chi)$ assigns is γ', and we update the memory state to be m'.

Problem Definition; Quantitative Shield Synthesis Consider an abstraction MDP $\mathcal{A}$, a behavioral score $\textsc{Beh}$, an interference score $\textsc{Int}$, both given as weighted automata, and a factor $\lambda \in [0, 1]$ with which we weigh the two scores. Our goal is to find an *optimal shield* w.r.t. these inputs as we define below. Consider a shield $\textsc{Shield}$ with memory set M. Let X be the set of abstract controllers. For $\textsc{Shield}$ and $\chi \in X$, let $\mathcal{D}(\textsc{Shield}, \chi)$ be the probability distribution over $A \times \Gamma \times \Gamma$ that the policy $pol(\textsc{Shield}, \chi)$ gives rise to. The *value* of $\textsc{Shield}$, denoted $val(\textsc{Shield})$, is $\sup_{\chi \in X} \mathbb{E}_{r \sim \mathcal{D}(\textsc{Shield}, \chi)}[\lambda \cdot \textsc{Int}(r) + (1 - \lambda) \cdot \textsc{Beh}(r)]$. An *optimal shield* is a shield whose value is $\inf_{\textsc{Shield}} val(\textsc{Shield})$.

Remark 1 **(Robustness and flexibility).** The problem definition we consider allows quantitative optimization of shields w.r.t. two dimensions of quantitative measures. Earlier works have considered shields but mainly with respect to Boolean measures in both dimensions. For example, in [30], shields for safety behavioral measures were constructed with a Boolean notion of interference, as well as a Boolean notion of shield correctness. In contrast we allow quantitative objectives in both dimensions which presents a much more general and robust framework. For example, the first measure of correctness can be quantitative and minimize the error rate, and the second measure can allow shields to correct but minimize the long-run average interference. Both of the above allows the shield to be flexible. Moreover, tuning the parameter λ allows flexible trade-off between the two.

We allow a robust class of quantitative specifications using weighted automata, which have been already established as a robust specification framework. Any automata model can be used in the framework, not necessarily the ones we use here. For example, weighted automata that discount the future or process only finite-words are suitable for planning purposes [32]. Thus our framework is a very robust and flexible framework for quantitative shield synthesis. □

2.3 Examples

In Remark 1 we already discussed the flexibility of the framework. We now present concrete examples of instantiations of the optimization problem above on our running example, which illustrate how quantitative shields can be used to cope with limitations of learned controllers.

Dealing with Unexpected Plant Behavior; Rush-Hour Traffic. Consider the abstraction described in Example 1, where each abstract state is a 4-dimensional vector that represents the number of waiting cars in each direction. The behavioral score we use is called the *max queue*. It charges an abstract state $a \in \{0, \ldots, k\}^4$ with the size of the maximal queue, no matter what the issued action is, thus $cost_{\text{BEH}}(a) = \max_{i \in \{1,2,3,4\}} a_i$. A shield that minimizes the max-queue cost will prioritize the direction with the largest queue. For the interference score, we use a score that we call the *basic* interference score; we charge the shield 1 whenever it changes the controller's action and otherwise we charge it 0, and take the long-run average of the costs. Recall that in the construction in Example 1, we chose uniformly at random the vector of incoming cars. Here, in order to model rush-hour traffic, we use a different distribution, where we let p_j be the probability that a car enters the j-th queue. Then, the probability of a vector $\langle i_1, i_2, i_3, i_4 \rangle \in \{0, 1\}^4$ is $\prod_{1 \leq j \leq 4}(p_j \cdot i_j + (1 - p_j) \cdot (1 - i_j))$. To model a higher load traveling on the North-South route, we increase p_1 and p_3 beyond 0.5.

Weighing Different Goals; Local Fairness. Suppose the controller is trained to maximize the number of cars passing a *city*. Thus, it aims to maximize the speed of the cars in the city and prioritizes highways over farm roads. A secondary objective for a controller is to minimize local queues. Rather than adding this objective in the training phase, which can have an un-expected outcome, we can add a local shield for each junction. To synthesize the shield, we use the same abstraction and basic interference score as in the above. The behavioral score we use charges an abstract state $a \in \{0, \ldots, k\}^4$ with difference $|(a_1 + a_3) - (a_2 + a_4)|$, thus the greater the inequality between the two waiting directions, the higher the cost.

Adding Features to the Controller; Prioritizing Public Transport. Suppose a controller is trained to increase throughput in a junction. After the controller is trained, a designer wants to add a functionality to the controller that prioritizes buses over personal vehicles. That is, if a bus is waiting in the North direction, and no bus is waiting in either the East or West directions, then the light should be North-South, and the other cases are similar. The abstraction we use is simpler than the ones above since we only differentiate between a case in which a bus is present or not, thus the abstract states are $\{0,1\}^4$, where the indices represent the directions clockwise starting from North. Let $\gamma = \text{NS}$. The behavioral cost of a state a is 1 when $a_2 = a_4 = 0$ and $a_1 = 1$ or $a_3 = 1$. The interference score we use is the basic one. A shield guarantees that in the long run, the specification is essentially never violated.

3 A Game-Theoretic Approach to Quantitative Shield Synthesis

In order to synthesize optimal shields we construct a two-player stochastic game [10], where we associate Player 2 with the shield and Player 1 with the controller. The game is defined on top of an abstraction and the players' objectives are given by the two performance measures. We first formally define stochastic games, then we construct the shield synthesis game, and finally show how to extract a shield from a strategy for Player 2.

Stochastic Graph Games. The game is played on a graph by placing a token on a vertex and letting the players move it throughout the graph. For ease of presentation, we fix the order in which the players move: first, Player 1, then Player 2, and then "Nature", i.e., the next vertex is chosen randomly. Edges have costs, which, again for convenience, appear only on edges following Player 2 moves. Formally, a two-player stochastic graph-game is $\langle V_1, V_2, V_N, E, \Pr, \mathit{cost}\rangle$, where $V = V_1 \cup V_2 \cup V_N$ is a finite set of vertices that is partitioned into three sets, for $i \in \{1,2\}$, Player i controls the vertices in V_i and "Nature" controls the vertices in V_N, $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_N)$ is a set of deterministic edges, $\Pr : V_N \times V_1 \to [0,1]$ is a probabilistic transition function, and $\mathit{cost} : (V_2 \times V_N) \to \mathbb{Q}$. Suppose the token reaches $v \in V$. If $v \in V_i$, for $i \in \{1,2\}$, then Player i chooses the next position of the token $u \in V$, such that $E(v,u)$. If $v \in V_N$, then the next position is chosen randomly; namely, the token moves to $u \in V$ with probability $\Pr[v,u]$.

The game is a *zero-sum game*; Player 1 tries to maximize the expected long-run average of the accumulated costs, and Player 2 tries to minimize it. A *strategy* for Player i, for $i \in \{1,2\}$, is a function that takes a history in $V^* \cdot V_i$ and returns the next vertex to move the token to. The games we consider admit *memoryless* optimal strategies, thus it suffices to define a Player i strategy as a function from V_i to V. We associate a *payoff* with two strategies f_1 and f_2, which we define next. Given f_1 and f_2, it is not hard to construct a Markov chain $\mathcal{M}$ with states V_N and with weights on the edges: for $v, u \in V_N$, the probability of moving from v to u in $\mathcal{M}$ is $\Pr_{\mathcal{M}}[v,u] = \sum_{w \in V_1 : f_2(f_1(w))=u} \Pr[v,w]$ and the cost of the edge is $\mathit{cost}_{\mathcal{M}}(v,u) = \sum_{w \in V_1 : f_2(f_1(w))=u} \Pr[v,w] \cdot \mathit{cost}(f_1(w), u)$. The *stationary distribution* s_v of a vertex $v \in V_N$ in $\mathcal{M}$ is a well known concept [43] and it intuitively measures the long-run average time that is spend in v. The payoff w.r.t. f_1 and f_2, denoted $\text{payoff}(f_1, f_2)$ is $\sum_{v,u \in V_N} s_v \cdot \Pr_{\mathcal{M}}[v,u] \cdot \mathit{cost}_{\mathcal{M}}(v,u)$. The payoff of a strategy is the payoff it guarantees against any strategy of the other player, thus $\text{payoff}(f_1) = \inf_{f_2} \text{payoff}(f_1, f_2)$. A strategy is *optimal* for Player 1 if it achieves the optimal payoff, thus f is optimal if $\text{payoff}(f) = \sup_{f_1} \text{payoff}(f_1)$. The definitions for Player 2 are dual.

Constructing the Synthesis Game. Consider an abstraction MDP $\mathcal{A} = \langle \Gamma, A, a_0, \delta \rangle$, weighted automata for the behavioral score $\textsc{Beh} = \langle A \times \Gamma, Q_{\textsc{Beh}}, q_0^{\textsc{Beh}}, \Delta_{\textsc{Beh}}, cost_{\textsc{Beh}} \rangle$ and interference score $\textsc{Int} = \langle \Gamma \times \Gamma, Q_{\textsc{Int}}, q_0^{\textsc{Int}}, \Delta_{\textsc{Int}}, cost_{\textsc{Int}} \rangle$, and a factor $\lambda \in [0, 1]$. We associate Player 1 with the controller and Player 2 with the shield. In each step, the controller first chooses an action, then the shield chooses whether to alter it, and the next state is selected at random. Let $S = A \times Q_{\textsc{Int}} \times Q_{\textsc{Beh}}$. We define $\mathcal{G}_{\mathcal{A},\textsc{Beh},\textsc{Int},\lambda} = \langle V_1, V_2, V_N, E, \Pr, cost \rangle$, where

- $V_1 = S$,
- $V_2 = S \times \Gamma$,
- $V_N = S \times \Gamma \times \{N\}$, where the purpose of N is to differentiate between the vertices,
- $E(s, \langle s, \gamma \rangle)$
  for $s \in S$ and $\gamma \in \Gamma$, and $E(\langle s, \gamma \rangle, \langle s', \gamma', N \rangle)$ for $s = \langle a, q_1, q_2 \rangle \in S, \gamma, \gamma' \in \Gamma$, and $s' = \langle a, q_1', q_2' \rangle \in S$ s.t. $\Delta_{\textsc{Int}}(q_1, \langle \gamma, \gamma' \rangle, q_1')$ and $\Delta_{\textsc{Beh}}(q_2, \langle a, \gamma' \rangle, q_2')$,
- $\Pr[\langle \langle a, q_1, q_2 \rangle, \gamma, N \rangle, \langle a', q_1, q_2 \rangle] = \delta(a, \gamma)(a')$, and
- for $s = \langle a, q_1, q_2 \rangle$ and $s' = \langle a, q_1', q_2' \rangle$ as in the above, we have $cost(\langle s, \gamma \rangle, \langle s', \gamma', N \rangle) = \lambda \cdot cost_{\textsc{Int}}(q_1, \langle \gamma, \gamma' \rangle, q_1') + (1 - \lambda) \cdot cost_{\textsc{Beh}}(q_2, \langle \gamma', a \rangle, q_2')$.

From Strategies to Shields. Recall that the game $\mathcal{G}_{\mathcal{A},\textsc{Beh},\textsc{Int},\lambda}$ admits memoryless optimal strategies. Consider an optimal memoryless strategy f for Player 2. Thus, given a Player 2 vertex in V_2, the function f returns a vertex in V_N to move to. The shield $\textsc{Shield}_f$ that is associated with f has the memory set $M = Q_{\textsc{Int}} \times Q_{\textsc{Beh}}$ and the initial memory state is $\langle q_0^{\textsc{Int}}, q_0^{\textsc{Beh}} \rangle$. Given an abstract state $a \in A$, a memory state $\langle q_{\textsc{Int}}, q_{\textsc{Beh}} \rangle \in M$, and a controller action $\gamma \in \Gamma$, let $\langle a, q_{\textsc{Int}}', q_{\textsc{Beh}}', \gamma' \rangle = f(a, q_{\textsc{Int}}, q_{\textsc{Beh}}, \gamma)$. The shield $\textsc{Shield}_f$ returns the action γ' and the updated memory state $\langle q_{\textsc{Int}}', q_{\textsc{Beh}}' \rangle$.

Theorem 1. *Given an abstraction $\mathcal{A}$, weighted automata* $\textsc{Beh}$ *and* $\textsc{Int}$*, and a factor λ, the game $\mathcal{G}_{\mathcal{A},\textsc{Beh},\textsc{Int},\lambda}$ admits optimal memoryless strategies. Let f be an optimal memoryless strategy for Player* 2*. The shield* $\textsc{Shield}_f$ *is an optimal shield w.r.t. $\mathcal{A}$,* $\textsc{Beh}$*,* $\textsc{Int}$*, and λ.*

Remark 2 **(Shield size).** Recall that a shield is a function $\textsc{Shield} : A \times \Gamma \times M \to \Gamma \times M$, which we store as a table. The *size* of the shield is the size of the domain, namely the number of entries in the table. Given an abstraction with n_1 states, a set of possible commands Γ, and weighted automata with n_2 and n_3 states, the size of the shield we construct is $n_1 \cdot n_2 \cdot n_3 \cdot |\Gamma|$. □

Remark 3. Our construction of the game can be seen as a two-step procedure: we construct a stochastic game with two mean-payoff objectives, a.k.a. a *two-dimensional* game, where the shield player's goal is to minimize both the behavioral and interference scores separately. We then reduce the game to a "one-dimension" game by weighing the scores with the parameter λ. We perform this reduction for several reasons. First, while multi-dimensional quantitative objectives have been studied in several cases, such as MDPs [4,6,7] and special problems of stochastic games (e.g., almost-sure winning) [2,5,8], there is no general algorithmic solution known for stochastic games with two-dimensional objectives. Second, even for non-stochastic games with two-dimensional quantitative objectives, infinite-memory is required in general [48]. Finally, in our setting, the parameter λ provides a meaningful tradeoff: it can be associated with how well we value the quality of the controller. If the controller is of poor quality, then we charge the shield less for interference and set λ to be low. On the other hand, for a high-quality controller, we charge the shield more for interferences and set a high value for λ. $\square$

4 Case Study

We experiment with our framework in designing quantitative shields for traffic-light controllers that are trained using reinforcement-learning (RL). We illustrate the usefulness of shields in dealing with limitations of RL as well as providing an intuitive framework to complement RL techniques.

Traffic Simulation. All experiments were conducted using traffic simulator "Simulation of Urban MObility" (SUMO, for short) [31] v0.22 using the SUMO Python API. Incoming traffic in the cities is chosen randomly. The simulations were executed on a desktop computer with a 4 x 2.70 GHz Intel Core i7-7500U CPU, 7.7 GB of RAM running Ubuntu 16.04.

The Traffic Light Controller. We use RL to train a city-wide traffic-signal controller. Intuitively, the controller is aware of the waiting cars in each junction and its actions constitute a light assignment to all the junctions. We train a controller using a deep convolutional Q-network [37]. In most of the networks we test with, there are two controlled junctions. The input vector to the neural network is a 16-dimensional vector, where 8 dimensions represent a junction. For each junction, the first four components state the number of cars approaching the junction and the last four components state the accumulated waiting time of the cars in each one of the lanes. For example, in Fig. 1, the first four components are $(3, 6, 3, 1)$, thus the controller's state is not trimmed at 5. The controller is trained to minimize both the number of cars waiting in the queues and the total waiting time. For each junction i, the controller can choose to set the light to be either NS_i or EW_i, thus the set of possible actions is $\Gamma = \{\text{NS}_1\text{NS}_2, \text{EW}_1\text{NS}_2, \text{NS}_1\text{EW}_2, \text{EW}_1\text{EW}_2\}$.

We use a network consisting of 4 layers: The input layer is a convolutional layer with 16 nodes, the first hidden and the second hidden layers consisting out of 604 nodes and 1166 nodes, respectively. The output layer consists of 4 neurons with linear activation functions, each representing one of the above mentioned actions listed in Γ. The Q-learning uses the learning rate $\alpha = 0.001$ and the discount factor 0.95 for the Q-update and an ϵ-greedy exploration policy. The artificial neural network is built on an open source implementation[1] using Keras [9] and additional optimized functionality was provided by the NumPy [40] library. We train for 100 training epochs, where each epoch is 1500 seconds of simulated traffic, plus 2000 additional seconds in which no new cars are introduced. The total training time of the agent is roughly 1.5 hours. While the RL procedure that we use is simple procedure, it is inspired by standard approaches

[1] https://github.com/Wert1996/Traffic-Optimisation.

to learning traffic controllers and produces controllers that perform relatively well also with no shield.

The Shield. We synthesize a "local" shield for a junction and copy the shield for each junction in the city. Recall that the first step in constructing the synthesis game is to construct an abstraction of the plant, which intuitively represents the information according to which the shield makes its decisions. The abstraction we use is described in Example 1; each state is a 4-dimensional integer in $\{0, \ldots, k\}$, which represents an abstraction of the number of waiting cars in each direction, cut-off by $k \in \mathbb{N}$. As elaborated in the example, when a shield assigns a green light to a direction, we evict a car from the two respectable queues, and select the incoming cars uniformly at random. Regarding objectives, in most of our experiments, the behavioral score we use charges an abstract state $a \in \{0, \ldots, k\}^4$ with $|(a_1 + a_3) - (a_2 + a_4)|$, thus the shield aims to balance the total number of waiting cars per direction. The interference score we use charges the shield 1 for altering the controller's action.

Since we use simple automata for objectives, the size of the shields we use is $|A \times \Gamma|$, where $|\Gamma| = 2$. In our experiments, we cut-off the queues at $k = 6$, which results in a shield of size 2592. The synthesis procedure's running time is in the order of minutes. We have already pointed out that we are interested in small light-weight shields, and this is indeed what we construct. In terms of absolute size, the shield takes $\sim$60 KB versus the controller who takes $\sim$3 MB; a difference of 2 orders of magnitude.

Our synthesis procedure includes a solution to a stochastic mean-payoff game. The complexity of solving such games is an interesting combinatorial problem in NP and coNP (thus unlikely to be NP-hard) for which the existence of a polynomial-time algorithm is major long-standing open problem. The current best-known algorithms are exponential, and even for special cases like turn-based deterministic mean-payoff games or turn-based stochastic games with reachability objectives, no polynomial-time algorithms are known. The algorithm we implemented is called the *strategy iteration* algorithm [22,23] in which one starts with a strategy and iteratively improves it, where each iteration requires polynomial time. While the algorithm's worst-case complexity is exponential, in practice, the algorithm has been widely observed to terminate in a few number of iterations.

Evaluating Performance. Throughout all our experiments, we use a unified and concrete measure of performance: the total waiting time of the cars in the city. Our assumption is that minimizing this measure is the main objective of the designer of the traffic light system for the city. While performance is part of the objective function when training the controller, the other components of the objective are used in order to improve training. Similarly, the behavioral measure we use when synthesizing shields is chosen heuristically in order to construct shields that improve concrete performance.

The Effect of Changing $\boldsymbol{\lambda}$. Recall that we use $\lambda \in [0, 1]$ in order to weigh between the behavioral and interference measures of a shield, where the larger λ is, the more the shield is charged for interference. In our first set of experiments, we fix all parameters apart from λ and synthesize shields for a city that has two controllable junctions. In the first experiment, we use a random traffic flow that is similar to the one used in training.

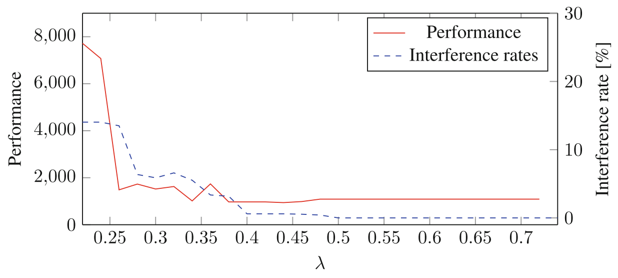


Fig. 2. Results for shields constructed with various λ values, together with a fixed plant and controller, where the simulation traffic distribution matches the one the controller is trained for.

We depict the results of the simulation in Fig. 2. We make several observations on the results below.

Interference. We observe that the ratio of the time that the shield intervenes is low: for most values of λ the ratio is well below 10%. For large values of λ, interference is too costly, and the shields become *trivial*, namely it never alters the actions of the controller. The performance we observe is thus the performance of the controller with no shield. In this set of experiments, we observe that the threshold after which shields become trivial is $\lambda = 0.5$, and for different setups, the threshold changes.

Performance. We observe that performance as function of λ, is a curve-like function. When λ is small, altering commands is cheap, the shield intervenes more frequently, and performance drops. This performance drop is expected: the shield is a simple device and the quality of its routing decisions cannot compete with the trained controller. This drop is also encouraging since it illustrates that our experimental setting is interesting. Surprisingly, we observe that the curve is in fact a paraboloid: for some values, e.g., $\lambda = 0.4$, the shield improves the performance of the controller. We find it unexpected that the shield improves performance even when observing trained behavior, and this performance increase is observed more significantly in the next experiments.

Rush-Hour Traffic. In Fig. 3, we use a shield to add robustness to a controller for behavior it was not trained for. We see a more significant performance gain in this exper-

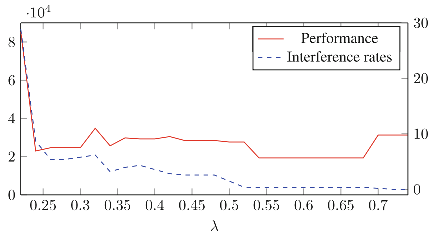


Fig. 3. Similar to Fig. 2 only that the simulation traffic distribution models rush-hour traffic.

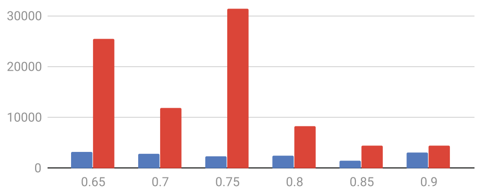


Fig. 4. Comparing the variability in performance of the different controllers, with shield (blue) and without a shield (red). (Color figure online)

iment. We use the controller from the previous experiment, which is trained for uniform car arrival. We simulate it in a network with "rush-hour" traffic, which we model by significantly increasing the traffic load in the North-South direction. We synthesize shields that prefer to evict traffic from the North-South queue over the East-West queue. We achieve this by altering the objective in the stochastic game; we charge the shield a greater penalty for cars waiting in these queues over the other queues. For most values of λ below 0.7, we see a performance gain. Note that the performance of the controller with no shield is depicted on the far right, where the shield is trivial. An alternative approach to synthesize a shield would be to alter the probabilities in the abstraction, but we found that altering the weights results in a better performance gain.

Reducing Variability. Machine learning techniques are intricate, require expertise, and a fine tuning of parameters. This set of experiments show how the use of shields reduces variability of the controllers, and as a result, it reduces the importance of choosing the optimal parameters in the training phase. We fix one of the shields from the first experiment with $\lambda = 0.4$. We observe performance in a city with various controllers, which are trained with varying training parameters, when the controllers are run with and without the shield and on various traffic conditions that sometimes differ from the ones they are trained on.

The city we experiment with consists of a main two-lane road that crosses the city from East to West. The main road has two junctions in which smaller "farm roads" meet the main road. We refer to the *bulk traffic* as the traffic that only "crosses the city"; namely, it flows only on the main road either from East to West or in the opposite direction. For $r \in [0, 1]$, Controller-r is trained where the ratio of the bulk traffic out of the total traffic is r. That is, the higher r is, the less traffic travels on the farm roads. We run simulations in which Controller-r observes bulk traffic $k \in [0, 1]$, which it was not necessarily trained for.

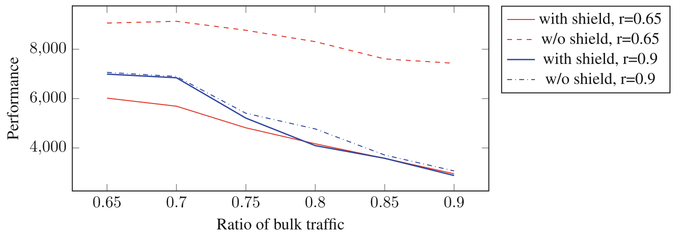


Fig. 5. Results for Controllers-0.65 and 0.9 exhibiting traffic that they are not trained for, with and without a shield. Performance is the total waiting time of the cars in the city.

In Fig. 5, we depict the performance of two controllers for various traffic settings. We observe, in these two controllers as well as the others, that operating with a shield consistently improves performance. The plots illustrate the unexpected behavior of machine-learning techniques: e.g., when run without a shield, Controller-0.9 outperforms Controller-0.65 in all settings, even in the setting 0.65 on which Controller-0.65 was trained on. Thus, a designer who expects a traffic flow of 0.65, would be better off training with a traffic of 0.9. A shield improves performance and thus reduces the importance of which training data to use.

Measuring Variability. In Fig. 4, we depict the variability in performance between the controllers. The higher the variability is, the more significant it is to choose the right parameters when training the controller. Formally, let $R = \{0.65, 0.7, 0.75, 0.8, 0.85, 0.9\}$. For $r, k \in R$, we let $\text{Perf}(r, k)$ denote the performance (total waiting times) when Controller-r observes bulk traffic k. For each $k \in R$, we depict $\max_{r \in R} \text{Perf}(r, k) - \min_{r' \in R} \text{Perf}(r', k)$, when operating with and without a shield.

Clearly, the variability with a shield is significantly lower than without one. This data shows that when operating with a shield, it does not make much difference if a designer trains a controller with setting r or r'. When operating without a shield, the difference is significant.

Overcoming Liveness Bugs. Finding bugs in learned controllers is a challenging task. Shields bypass the need to find bugs since they treat the controller as a black-box and correct its behavior. We illustrate their usefulness in dealing with liveness bugs. In the same network as in the previous setting, we experiment with a controller whose training process lacked variability. In Fig. 6, we depict the light configuration throughout the experiment on the main road; the horizontal axis represents time, red means a red light for the main road and dually green. Initially, the controller performs well, but roughly half-way through the simulation it hits a bad state after which the light stays red. The shield, with only a few interferences, which are represented with dots, manages to recover the controller from its stuck state. In Fig. 7, we depict the number of waiting cars in the city, which clearly skyrockets once the controller gets stuck. It is evident that initially, the controller performs well. This point highlights that it is difficult to recog-

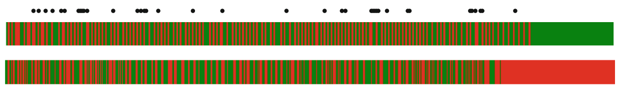


Fig. 6. The light in the East-West direction (the main road) of a junction. On bottom, with no shield the controller is stuck. On top, the shield's interferences are marked with dots.

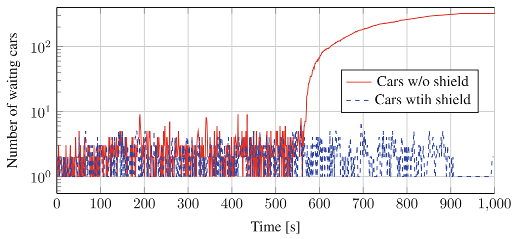


Fig. 7. The total number of waiting cars (log-scale) with and without a shield. Initially, the controller performs well on its own, until it gets stuck and traffic in the city freezes.

nize when a controller has a bug – in order to catch such a bug, a designer would need to find the right simulation and run it half way through before the bug appears.

One way to regain liveness would be to synthesize a shield for the qualitative property "each direction eventually gets a green light". Instead, we use a shield that is synthesized for the quantitative specification as in the previous experiment. The shield, with a total of only 20 alterations is able to recover the controller from the bad state it is stuck in, and traffic flows correctly.

Adding Functionality; Prioritizing Public Transport. Learned controllers are monolithic. Adding functionality to a controller requires a complete re-training, which is time consuming, computationally costly, and requires care; changes in the objective can cause unexpected side effects to the performance. We illustrate how, using a shield, we can add to an existing controller, the functionality of prioritizing public transport.

The abstraction over which the shield is constructed slightly differs from the one used in the other experiments. The abstract state space is the same, namely four-dimensional vectors, though we interpret the entries as the positions of a bus in the respective queue. For example, the state $(0, 3, 0, 1)$ represents no bus in the North queue and a bus which is waiting, third in line, in the East queue. Outgoing edges from an abstract state also differ as they take into account, using probability, that vehicles might enter the queues between buses. For the behavioral score, we charge an abstract state with the sum of its entries, thus the shield is charged whenever buses are waiting and it aims to evict them from the queues as soon as possible.

In Fig. 8, we depict the performance of all vehicles and only buses as a function of the weighing factor λ. The result of this experiment is positive; the predicted behavior is observed. Indeed, when λ is small, interferences are cheap, which increase bus performance at the expense of the general performance. The experiment illustrates that the parameter λ is a convenient method to control the degree of prioritization of buses.

Local Fairness. In this experiment, we add local fairness to a controller that was trained for a global objective. We experiment with a network with four junctions and a city-wide controller, which aims to minimize total waiting times. Figure 9 shows that when the controller is deployed on its own, queues form in the city whereas a shield, which was synthesized as in the first experiments, prevents such local queues from forming.

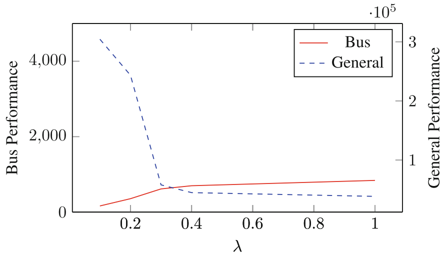


Fig. 8. The waiting time of buses/all vehicles with shields parameterized by λ.

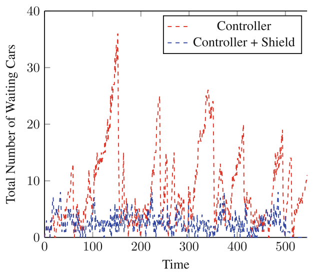


Fig. 9. Comparing the amount of waiting cars with and without a shield.

5 Discussion and Future Work

We suggest a framework for automatically synthesizing quantitative runtime shields to cope with limitations of machine-learning techniques. We show how shields can increase robustness to untrained behavior, deal with liveness bugs without verification, add features without retraining, and decrease variability of performance due to changes in the training parameters, which is especially helpful for machine learning non-experts. We use weighted automata to evaluate controller and shield behavior and construct a game whose solution is an optimal shield w.r.t. a weighted specification and a plant abstraction. The framework is robust and can be applied in any setting where learned or other black-box controllers are used.

We list several directions for further research. In this work, we make no assumptions on the controller and treat it adversarially. Since the controller might have bugs, modelling it as adversarial is reasonable. Though, it is also a crude abstraction since typically, the objectives of the controller and shield are similar. For future work, we plan to study ways to model the spectrum between cooperative and adversarial controllers together with solution concepts for the games that they give rise to.

In this work we make no assumptions on the relationship between the plant and the abstraction. While the constructed shields are optimal w.r.t. the given abstraction, the scores they guarantee w.r.t. the abstraction do not imply performance guarantees on the plant. To be able to produce performance guarantees on the concrete plant, we need guarantees on the relationship between the plant its abstraction. For future work, we plan to study the addition of such guarantees and how they affect the quality measures.

References

1. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: AAAI. AAAI Press (2018)
2. Basset, N., Kwiatkowska, M.Z., Wiltsche, C.: Compositional strategy synthesis for stochastic games with multiple objectives. Inf. Comput. **261**(Part), 536–587 (2018)
3. Bloem, R., Chatterjee, K., Jobstmann, B.: Graph games and reactive synthesis. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 921–962. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_27
4. Brázdil, T., Brozek, V., Chatterjee, K., Forejt, V., Kucera, A.: Two views on multiple mean-payoff objectives in Markov decision processes. Log. Methods Comput. Sci. **10**(1) (2014). https://lmcs.episciences.org/1156
5. Chatterjee, K., Doyen, L.: Perfect-information stochastic games with generalized mean-payoff objectives. In: Proceedings of the 31st LICS, pp. 247–256 (2016)
6. Chatterjee, K., Kretínská, Z., Kretínský, J.: Unifying two views on multiple mean-payoff objectives in Markov decision processes. Log. Methods Comput. Sci. **13**(2) (2017). https://lmcs.episciences.org/3757
7. Chatterjee, K., Majumdar, R., Henzinger, T. A.: Markov decision processes with multiple objectives. In: Proceedings of the 23rd STACS, pp. 325–336 (2006)
8. Chen, T., Forejt, V., Kwiatkowska, M. Z., Simaitis, A., Trivedi, A., Ummels, M.: Playing stochastic games precisely. In: Proceedings of the 23rd CONCUR, pp. 348–363 (2012)
9. Chollet, F.: keras (2015). https://github.com/fchollet/keras

10. Condon, A.: On algorithms for simple stochastic games. In: Proceedings of the DIMACS, pp. 51–72 (1990)
11. Dalal, G., Dvijotham, K., Vecerik, M., Hester, T., Paduraru, C., Tassa, Y.: Safe exploration in continuous action spaces. coRR, abs/1801.08757 (2017). arXiv:1801.08757
12. Desai, A., Ghosh, S., Seshia, S. A., Shankar, N., Tiwari, A.: SOTER: programming safe robotics system using runtime assurance. coRR, abs/1808.07921 (2018). arXiv:1808.07921
13. Dewey, D.: Reinforcement learning and the reward engineering principle. In: 2014 AAAI Spring Symposium Series (2014)
14. Droste, M., Kuich, W., Vogler, H.: Handbook of Weighted Automata. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01492-5
15. Duan, Y., Chen, X., Houthooft, R., Schulman, J., Abbeel, P.: Benchmarking deep reinforcement learning for continuous control. In: Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, 19–24 June 2016, pp. 1329–1338 (2016)
16. Falcone, Y., Mounier, L., Fernandez, J.-C., Richier, J.-L.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods Syst. Des. **38**(3), 223–262 (2011)
17. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods: toward safe control through proof and learning. In: AAAI. AAAI Press (2018)
18. García, J., Fernández, F.: A comprehensive survey on safe reinforcement learning. J. Mach. Learn. Res. **16**, 1437–1480 (2015)
19. Geibel, P.: Reinforcement learning for MDPs with constraints. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 646–653. Springer, Heidelberg (2006). https://doi.org/10.1007/11871842_63
20. Genders, W., Razavi, S.: Asynchronous n-step q-learning adaptive traffic signal control. J. Intell. Trans. Syst. **23**(4), 319–331 (2019)
21. Hamlen, K.W., Morrisett, J.G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Trans. Program. Lang. Syst. **28**(1), 175–205 (2006)
22. Hoffman, A.J., Karp, R.M.: On nonterminating stochastic games. Manag. Sci. **12**(5), 359–370 (1966)
23. Howard, A.R.: Dynamic Programming and Markov Processes. MIT Press, Cambridge (1960)
24. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Proceedings of the 29th CAV, pp. 3–29 (2017)
25. Jansen, N., Könighofer, B., Junges, S., Bloem, R.: Shielded decision-making in MDPs. CoRR, arXiv:1807.06096 (2018)
26. Ji, Y., Lafortune, S.: Enforcing opacity by publicly known edit functions. In: 56th IEEE Annual Conference on Decision and Control, CDC 2017, Melbourne, Australia, 12–15 December 2017, pp. 4866–4871 (2017)
27. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artif. Intell. **101**(1), 99–134 (1998)
28. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: a survey. JAIR **4**, 237–285 (1996)
29. Katz, G., Barrett, C.W., Dill, C. W., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Proceedings of the 29th CAV, pp. 97–117 (2017)
30. Könighofer, B., Alshiekh, M., Bloem, R., Humphrey, L., Könighofer, R., Topcu, U., Wang, C.: Shield synthesis. Formal Methods Syst. Des. **51**(2), 332–361 (2017)
31. Krajzewicz, D., Erdmann, J., Behrisch, M., Bieker, L.: Recent development and applications of SUMO - Simulation of Urban MObility. Int. J. Adv. Syst. Meas. **5**(3&4), 128–138 (2012)

32. Lahijanian, M., Almagor, S., Fried, D., Kavraki, L.E., Vardi, M.Y.: This time the robot settles for a cost: a quantitative approach to temporal logic planning with partial satisfaction. In: Proceedings of the 29th AAAI, pp. 3664–3671 (2015)
33. Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J., Quillen, D.: Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. I. J. Robot. Res. **37**(4–5), 421–436 (2018)
34. Lillicrap, T.P.: Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971 (2015)
35. Mannion, P., Duggan, J., Howley, E.: An experimental review of reinforcement learning algorithms for adaptive traffic signal control. In: McCluskey, T.L., Kotsialos, A., Müller, J.P., Klügl, F., Rana, O., Schumann, R. (eds.) Autonomic Road Transport Support Systems. AS, pp. 47–66. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-25808-9_4
36. Mason, G., Calinescu, R., Kudenko, D., Banks, A.: Assured reinforcement learning with formally verified abstract policies. In: Proceedings of the 9th International Conference on Agents and Artificial Intelligence, ICAART 2017, Porto, Portugal, 24–26 February 2017, vol. 2, pp. 105–117 (2017)
37. Mnih, V.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
38. Moldovan, T.M., Abbeel, P.: Safe exploration in Markov decision processes. In: Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 – July 1, 2012 (2012)
39. Mousavi, S.S., Schukat, M., Howley, E.: Traffic light control using deep policy-gradient and value-function-based reinforcement learning. IET Intell. Trans. Syst. **11**(7), 417–423 (2017)
40. Oliphant, T.E.: Guide to NumPy, 2nd edn. CreateSpace Independent Publishing Platform, USA (2015)
41. Phan, D., Yang, J., Grosu, R., Smolka, S.A., Stoller, S.D.: Collision avoidance for mobile robots with limited sensing and limited information about moving obstacles. Formal Methods Syst. Des. **51**(1), 62–86 (2017)
42. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, 11–13 January 1989, pp. 179–190 (1989)
43. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons Inc., New York (2005)
44. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete-event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987)
45. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1), 30–50 (2000)
46. Seshia, S. A., Sadigh, D.: Towards verified artificial intelligence. CoRR, arXiv:1606.08514 (2016)
47. Sha, L.: Using simplicity to control complexity. IEEE Soft. **18**(4), 20–28 (2001)
48. Velner, Y., Chatterjee, K., Doyen, L., Henzinger, T.A., Rabinovich, A.M., Raskin, J.-F.: The complexity of multi-mean-payoff and multi-energy games. Inf. Comput. **241**, 177–196 (2015)
49. Wu, Y., Raman, V., Rawlings, B.C., Lafortune, S., Seshia, S.A.: Synthesis of obfuscation policies to ensure privacy and utility. J. Autom. Reasoning **60**(1), 107–131 (2018)