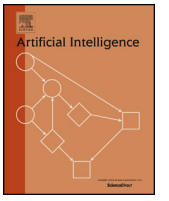
Artificial Intelligence 324 (2023) 103987

Contents lists available at ScienceDirect

Artificial Intelligence

journal homepage: www.elsevier.com/locate/artint



Risk-aware shielding of Partially Observable Monte Carlo Planning policies ☆

Giulio Mazzi, Alberto Castellini *, Alessandro Farinelli

Universita degli Studi di Verona, Dipartimento di Informatica, Strada Le Grazie 15, Verona, 37134, Italy

ARTICLE INFO

Article history:
Received 15 June 2022
Received in revised form 21 July 2023
Accepted 24 July 2023
Available online 9 August 2023

Keywords:
POMDP
POMCP
SMT
Risk-awareness
Shielding

ABSTRACT

Partially Observable Monte Carlo Planning (POMCP) is a powerful online algorithm that can generate approximate policies for large Partially Observable Markov Decision Processes. The online nature of this method supports scalability by avoiding complete policy representation. However, the lack of an explicit policy representation hinders interpretability and a proper evaluation of the risks an agent may incur. In this work, we propose a methodology based on Maximum Satisfiability Modulo Theory (MAX-SMT) for analyzing POMCP policies by inspecting their traces, namely, sequences of belief-action pairs generated by the algorithm. The proposed method explores local properties of the policy to build a compact and informative summary of the policy behaviour. Moreover, we introduce a rich and formal language that a domain expert can use to describe the expected behaviour of a policy. In more detail, we present a formulation that directly computes the risk involved in taking actions by considering the high-level elements specified by the expert. The final formula can identify risky decisions taken by POMCP that violate the expert indications. We show that this identification process can be used offline (to improve the policy's explainability and identify anomalous behaviours) or online (to shield the risky decisions of the POMCP algorithm). We present an extended evaluation of our approach on four domains: the well-known *tiger* and *rocksample* benchmarks, a problem of velocity regulation in mobile robots, and a problem of battery management in mobile robots. We test the methodology against a state-of-the-art anomaly detection algorithm to show that our approach can be used to identify anomalous behaviours in faulty POMCP. We also show, comparing the performance of shielded and unshielded POMCP, that the shielding mechanism can improve the system's performance. We provide an open-source implementation of the proposed methodologies at https://github.com/GiuMaz/XPOMCP.

1. Introduction

Planning in partially observable sequential environments is an important problem in artificial intelligence, with applications in robotics, cyber-physical system control, and other domains. The progressive spread of these systems in our daily life poses hard challenges related to safety. In particular, several domains in which autonomous agents perform critical tasks

☆ This paper is part of the Special Issue: "Risk-aware Autonomous Systems: Theory and Practice".

* Corresponding author.
E-mail address: alberto.castellini@univr.it (A. Castellini).

https://doi.org/10.1016/j.artint.2023.103987

(e.g., collaborative manufacturing, assisted driving, and so forth) are characterized by strong complexity and uncertainty, making safety preservation a difficult task.

Partially Observable Markov Decision Processes (POMDPs) [1] are one of the most popular frameworks for modelling decision making under uncertainty. POMDPs encode dynamical systems having partially observable states, where the hidden part of the state can only be estimated from observations with some degree of uncertainty. Computing exact optimal policies for POMDPs is very complex and unfeasible for large state spaces. Specifically, the problem is PSPACE-hard [2] when the horizon is finite and undecidable otherwise. Because of the importance of this framework, several approximate and online methods have been proposed to handle real-world instances. A pioneering algorithm for this purpose is Partially Observable Monte Carlo Planning (POMCP) [3] which uses Monte Carlo Tree Search (MCTS) [4] to compute the policy online. The approach is sampling-based and only computes the policy for the specific states the agent reaches while operating. POMCP uses a particle filter to represent the probability of each (hidden) state, called belief, given the history of observations.

A key problem of POMCP is understanding reasons and risks related to selecting specific actions. This happens because *i)* the policy for each belief is not available until the agent reaches that specific belief (and it is unfeasible to compute the policy for all beliefs reachable by the agent because they are infinite), *ii)* the policy is computed in a simulation-based way using a MCTS strategy which is difficult to interpret. The lack of a compact representation for POMCP policies makes evaluating and controlling the risks involved in using them difficult.

However, in several contexts, such as autonomous vehicles or collaborative robotics, humans need to understand why specific decisions are taken by autonomous agents employing complex decision strategies. The branch of AI that concerns the study and development of explainable AI systems is known as *explainable AI (XAI)* [5]. In particular, *explainable planning (XAIP)* [6,7] focuses on the explainability of planning methods. Wrong or unexpected behaviours of policies automatically synthesized by planning algorithms can have disastrous impacts, and detecting policy flaws is both challenging and fundamental for the safe deployment of autonomous agents. This is particularly important for approximated and online policies, such as those based on POMCP, which can scale to real-world problems but are also intrinsically hard to interpret and test.

In this work, we propose a methodology called *XPOMCP* for generating rule-based descriptions of POMCP policies with the end goal of evaluating and shielding risky actions. In more detail, experts provide qualitative information about expected system behaviours in the form of logical formulas, and XPOMCP returns quantitative details about those behaviours based on evidence observed in belief-action traces previously generated by the policy. For instance, in a logistic application for smart-manufacturing [8], an expert could define a logical rule saying that "the robot should move fast if it is highly confident that the aisle in which it is moving is not cluttered". After analyzing some traces produced by the policy, XPOMCP could explain that the robot usually moves fast if the probability of being in a cluttered segment is lower than 7%. This provides new knowledge about the risk involved in the decision-making process performed by the policy. This formula can be used as a base to identify anomalous behaviours in the policy (i.e., a behaviour that violates an expert's expectations). This is useful for identifying potential flaws in the agent's behaviour. To exploit this capability, we present a shielding mechanism that can be integrated directly into POMCP to prevent anomalous behaviour during the execution. This mechanism preemptively blocks (or *shields*) actions that are not allowed by the precomputed rules, and thus POMCP is free to select the best action among the acceptable ones.

To create a rule, XPOMCP asks the expert to define a *rule template* representing a question about the policy strategy. This is a set of logical formulas expressing partially defined assumptions. Parameters of rule templates are then computed from traces by a Satisfiability Modulo Theory (SMT) solver [9]. In particular, we formalize the parameter computation problem as a *MAX-SMT* problem. This allows us to write logical formulas using an expressive formalism and to compute optimal assignments also when the template is not fully satisfiable, which is very common in the analysis of real policies. This is a key and novel element of the proposed approach. It allows it to generate rules about the correct behaviour of the policy also from traces that contain errors since unsatisfied samples can be explicitly represented in the MAX-SMT framework. To facilitate the usage of our methodology, we also introduce a rich and flexible language, based on SMT-LIB [10], that allows defining rule templates.

By formulating a rule with a hard requirement, these descriptions can be used to ensure that a specification is satisfied at runtime. In this article, we present a particularly relevant use case for this approach, called *risk-aware shielding*. We define the *risk* as the probability of reaching undesired outcomes given the current belief and our choice of actions. In this formulation, the rule explicitly computes the risk involved in taking actions by exploiting the high-level descriptions provided by XPOMCP. The shield checks at runtime that this risk is always below a predefined threshold. Thus, the risk of undesired outcomes is explicitly limited, and this limitation can be expressed using high-level concepts instead of directly manipulating the POMCP algorithm. For example, we can extend a rule template for the smart manufacturing environment discussed above: "the robot should move fast if it is highly confident that the aisle in which it is moving is not cluttered, and the risk of colliding should never be higher than 5%". In this case, XPOMCP computes a rule that complies with the requirement while being as close as possible to the real POMCP policy.

We empirically evaluate the proposed method in an extended experimental setting composed of four benchmark domains for POMDPs: the well-known tiger and rocksample benchmarks, a problem of velocity regulation in mobile robot navigation, and a battery management problem for a mobile robot. To evaluate the ability of XPOMCP to detect anomalous actions, we introduce bugs in the POMCP algorithm. In the first case, we set a wrong exploration-exploitation constant c in the Upper Confidence bound for Trees (UCT) method (see [4] for details). This parameter has a very subtle effect on the decisions taken by POMCP because a wrong c parameter produces a non-deterministic imbalance of the MCTS used to evaluate action

values, which generates wrong decisions from time to time. This is a realistic case study for the wrong usage of POMCP that introduces risky decisions in the policy (i.e., POMCP underestimates the risk involved in its strategy). This error is challenging to identify and common in practice since the parameter c must be hand-tuned. However, certain domains use ad-hoc heuristics to reduce the impact of c and to improve performance. In these cases, changing the value of c has a limited effect. Thus, we also introduce a bug by reducing the number of particles/simulations n_p (usually, the number of particles and simulations coincides in POMCP) used to represent the belief. Using a reduced number of particles is desirable since the n_p is directly tied to the number of simulations that must be performed. However, using a limited number of simulations leads to wrong (hence risky) decisions. The analysis of this error is, therefore, also interesting. This setting is used, in particular, to evaluate the proposed method on *rocksample*. In our test, we compare the capability to identify anomalous actions of XPOMCP with that of *Isolation Forest* [11], a state-of-the-art anomaly detection algorithm. Specifically, our goal is to show that XPOMCP can incorporate prior knowledge and that using this knowledge allows XPOMCP to outperform isolation forest in identifying unexpected decisions. Moreover, we test the proposed shielding mechanism by integrating an explicit computation of the risks involved in a policy (i.e., risk-aware shields). We show that these shields can prevent unwanted outcomes, leading to better performance (i.e., POMCP achieves a higher cumulative reward thanks to the shield).

In summary, we provide the following contributions to the state-of-the-art:

- we generate a logic-based representation of POMCP policies which merges expert knowledge with observed belief-action traces and allows to describe the risk of the policy, detect anomalous actions and block them;
- we introduce a shielding procedure for POMCP, based on the logic-based rules, which prevents the planner from selecting anomalous and risky actions;
- we empirically evaluate the performance of the proposed method and its capability to improve performance and risk awareness in an extensive experimental setting concerning four domains, namely, *tiger*, *rocksample*, a problem of *velocity regulation* in mobile robots, and a problem of *battery management* in mobile robots.

The MAX-SMT based methodology for identifying unexpected decisions in POMCP policies has been presented for the first time in [12]. The shielding mechanism was presented, instead, for the first time in [13]. This manuscript significantly extends the two papers where the methodologies were first introduced. In particular, *i)* we define a formal language to express questions humans can ask about the properties of the policies they want to investigate and guarantee; *ii)* we present a new type of logic rules for our approach, namely *risk-aware rules*, for the shielding mechanism that combines policy descriptions with safety requirements; *iii)* we include new experiments on the *rocksample* domain (a popular benchmark) and the *battery* domain (that requires a shield reasoning on the effect of actions several steps ahead of the current belief); *iv)* we provide a more detailed explanation of the methodology, a deeper analysis of the empirical results and a thorough analysis of the state-of-the-art.

The rest of the paper is organized as follows: in Section 2 we describe the main differences between the proposed approach and related works in the literature, and we provide background knowledge on POMDPs, POMCP, and SMT. Section 3 provides a formalization of the methodology. In Section 4 we present an empirical evaluation of the methodology on four domains, and we discuss the results. Finally, Section 5 draws conclusions and sketches directions for future work.

2. Background and related work

In this section, we analyze the main research areas that are related to our work, namely, POMCP-based planning, safety and risk awareness for AI systems, shielding, and explainable planning. We also provide mathematical notation used in the rest of the manuscript.

2.1. Planning with POMCP

A Partially Observable Markov Decision Process (POMDP) [1,14] is a tuple $(S, A, O, T, Z, R, \gamma)$, where S is a set of partially observable *states*, A is a set of *actions*, Z is a finite set of *observations*, $T\colon S \times A \to \Pi(S)$ is the *state-transition model* (with $\Pi(s)$ space of probability distributions over states), $O\colon S \times A \to \Pi(Z)$ is the *observation model* (with $\Pi(Z)$ space of probability distributions over observations), $R\colon S \times A \to \mathbb{R}$ is the *reward function* and $\gamma \in [0, 1]$ is a *discount factor*. An agent must maximize the *discounted return* $E[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)]$. A probability distribution over states, called *belief*, is used to represent the partial observability of the true state. To solve a POMDP it is required to find a *policy*, namely a function $\pi\colon B \to A$ that maps beliefs (set B represents the belief-space) into actions.

We use Partially Observable Monte Carlo Planning (POMCP) to find a policy for a POMDP. POMCP was first introduced in 2010 [3] as an extension of the UCT algorithm [4] to partially observable environments. The strength of POMCP is that it does not require an explicit definition of the transition model (T), observation model (O), and reward (R). Instead, to represent the environment, it uses a *black-box simulator* taking a state and an action and returning the next state and the reward. POMCP uses a *Monte Carlo Tree Search* (MCTS) [15] at each time step to explore the belief space and select the best action. *Upper Confidence Bound for Trees (UCT)* [4] is used as a search strategy to select the subtrees to explore and balance exploration and exploitation. The belief is implemented as a *particle filter*, which is a sampling over the possible states updated at every step. At each time step, a particle is selected from the filter. Each particle represents a state. This state is

used as an initial point to perform a simulation in the Monte Carlo tree. Each simulation is a sequence of action-observation pairs which collects a discounted return. According to the backtracking procedure, the expected value of each action is then approximated by averaging the discounted returns of all simulations starting with that action. The particle filter is updated after performing a step in the real environment and receiving an observation. If required, new particles can be generated from the current state through a particle reinvigoration procedure.

Several extensions of POMCP have been realized since then. In particular, BA-POMCP [16] extends POMCP to Bayesian Adaptive POMDPs, allowing the model of the environment to be learned during execution; a version of POMCP for scalable planning in multiagent POMDPs [17]; a scalable extension of POMCP for dealing with cost constraints [18]; another version dealing with the introduction of prior knowledge on state-variable relationships [19]. Regarding applications of POMCP, some examples come from the exploration of partially known environments with robots [20] and active visual search in indoor environments [21,22], but also other applications are available in the literature [23]. MCTS-based approaches have been recently used also for developing agents with superhuman performance in the game of Go [24,25].

In our work, we propose a symbolic approach based on logic rules for explaining the (non-symbolic) POMCP policies, which are difficult to explain because they are generated online by a complex stochastic process based on MCTS. Moreover, we extend POMCP by integrating the rules generated by XPOMCP as a shielding mechanism.

2.2. Safety and risk-awareness for AI systems

Safety of artificial intelligence methods has recently become a central research topic [26–28]. Thus, several logic-based approaches have been proposed. In this work, we use propositional logic and the theory of linear real arithmetic to encode the rules that describe policies' behaviour. The problem of reasoning on the satisfiability of formulas involving propositional logic and first-order theories is called *Satisfiability Modulo Theory* (SMT). Specifically, we encode our formulas as a MAX-SMT [29] problem, which has two kinds of clauses, namely, *hard*, that must be satisfied, and *soft* that can be satisfied. A model of the MAX-SMT problem, hence, satisfies all the hard clauses and as many soft clauses as possible, and it is unsatisfiable only if no assignment satisfies all the hard clauses. Our rules must describe as many decisions as possible among those taken by the policy and MAX-SMT is a perfect formalism to reach this goal.

Several logic-based approaches have been developed to verify the safety of neural networks [30–37], motivated by the need to use these tools in safety-critical applications, such as autonomous driving. These methodologies mainly attempt to solve the verification problem on neural networks, which are highly non-linear because of the activation functions employed. Hence, standard verification methods are not applicable. The approaches cited above then adapt verification methods to work on specific classes of neural networks or approximate neural networks to allow the applicability of standard verification methods. In both cases, safety requirements must be mathematically specified in advance by logical formulas. Verification methodologies confirm that a network satisfies those requirements or they provide counterexamples. Our methodology differs from these methodologies because we use logic to compute a compact representation of the policy. Then, we use this representation to build a shield that blocks undesired actions at runtime. This is advantageous because it allows us to scale to large problems. We also present the safety requirement as a quantified risk involved in selecting an action, and we use this representation to block decisions that could lead to an unwanted state if the risk is considered unacceptable by the human expert.

The literature also provides various approaches specifically focused on synthesizing safe policies and verifying safety properties in policies for reinforcement learning and POMDP using an SMT-based approach. In these cases, temporal logic is often used to represent safety requirements since the involved models are sequential. SMT-based approaches are used, for instance, in [38–42]. These frameworks use formal approaches to synthesize a policy that satisfies predefined properties, an approach that presents scalability problems and has never been applied to online and approximate solvers such as POMCP. In contrast, we use a MAX-SMT solver to generate a compact representation of pre-computed policies that summarizes properties of interest. This enhances policy explainability without altering the policy itself. Thus, it can analyze complex policies generated online and deal with very large state spaces. This explanation can then be used to build a shielding mechanism that blocks decisions that do not respect the desired properties. To the best of our knowledge, there is no approach equivalent to XPOMCP in the literature that can build a mechanism that checks when high-level properties hold on POMCP.

We notice that our problem differs from that of *safe reinforcement learning* [43–45], in which policies must be generated without any knowledge of the environment and avoid risky exploration. In our context, instead, a black-box simulator of the environment is available (we deal with a planning problem according to [46]), and we can exploit it to build risk-aware rules (i.e., rules that directly quantify the risk involved in selecting actions).

Regarding specific methodologies for synthesizing POMDP policies, the approach presented in [47] can be used to satisfy safety requirements. In particular, the approach uses a SAT-solver to compute *winning regions*, which are a subset of beliefs for which it exists a policy that never reaches a forbidden state. This differs from our approach because our shields work with probabilistic constraints. For example, it is possible to accept an action if it has a small (but non-zero) chance of leading toward a forbidden state. This allows us to consider problems in which risk is unavoidable, but we are interested in risk-aware policies in which the risk is explicitly identified and kept under control.

Other methodologies formalize safety constraints using probabilistic approaches. In particular, [48] presents *Chance-Constrained POMDPs*, in which the execution risk of a policy is modelled as a bound on the probability of reaching states (in

a finite-horizon sequence of states) that violate a safety requirement. These constraints are considered by the Risk-bounded AO* (RAO*) algorithm to generate optimal deterministic policies that satisfy them. The risk constraints defined in XPOMCP have a different form. They are logical rules that relate sub-parts (i.e., subspaces) of the belief space to actions that can/cannot be performed in such sub-spaces. Therefore, we do not consider the probability to violate a state constraint in a sequence of states but the probability to go into a risky state performing a certain action from the current belief. Furthermore, the logical rules (considering the risk to go to unsafe states) are used by XPOMCP to activate/deactivate actions (shielding) during the Monte Carlo simulation process that generates online the policy, which is different from what RAO* does to generate offline the constrained policy. The relationships between the two strategies are not trivial and their analysis goes beyond the scope of this paper, however we consider this topic of interest for future investigations. Another approach for verifying properties in probabilistic systems is statistical model checking (SMC) [49]. This approach is applied to POMCP in [50] where SMC is used to verify that qualitative objectives, specified on possible states of the environment (i.e., safe reachability), are satisfied with a certain confidence level. These methodologies can be used to model and bound the risk involved in taking specific actions, as in XPOMCP, but these methods also differ from our approach because they specify properties (i.e., constraints) on states instead of beliefs. Moreover, we use the constraints to build a shield that blocks undesired behaviours. The methodology proposed in [50] also requires a large number of particles to generate reliable results, while our methodology performs well even with a limited number of particles, as shown in Section 4.4.2.

The growing number of autonomous systems deployed in safety-critical applications also leads to an increasing need to understand the risks involved in executing AI-generated policies. A system that quantifies these risks is known as *risk-aware autonomous systems*. In [51], the authors consider the problem of risk-aware planning for safe driving. They compute the risk of crashing on the street by analyzing previously collected data on the number of cars and crashes reported on that street. They propose an ad-hoc solution to the problem of finding the safest path by using graph algorithms. This differs from our approach for two reasons. First, we use previously collected data to build a compact policy representation and not to quantify risk. Instead, we quantify the risk as a function of the information provided by the black-box simulator used by POMCP. Second, we do not pre-compute a safe policy. Instead, we compute a shield that blocks risky action, and we use POMCP to compute the policy online by only considering legal (i.e., non-risky) actions.

2.3. Shielding

Shields have been used to verify critical properties of complex hardware designs in cases where traditional verification methods are unusable because of scalability issues [52,53]. A shielding mechanism for reinforcement learning agents is presented in [54]. This work presents two kinds of shielding mechanisms: a preemptive shield that selects a priori to which actions are allowed and a post-posed shield that intervenes only after the reinforcement learning algorithm selects an action. These shields enforce safety constraints expressed in temporal logic. An extension of this work is presented in [8] where the shield is specialized to prevent unsafe actions due to the agent's exploration of reinforcement learning-based policies. Our shielding mechanism, firstly introduced in [13], works as a preemptive shield, but our methodology has two important differences concerning the methodology presented above. Namely, we focus on partially observable problems. Thus our approach must deal with uncertainty, and we build the shield from a high-level representation. Specifically, the user does not have to specify the details of the shielding mechanism because XPOMCP computes them by analyzing execution traces.

The approach presented in [55] also uses a shielding mechanism based on a simplified representation of a policy. The methodology verifies properties related to the safety of a fully observable system modelled by Markov Decision Processes (MDPs). The approach works on a pre-trained neural network representing a black-box policy. It is composed of three main stages: *synthesis*, in which the complex neural policy is approximated using linear formulas that behave as close as possible to the original neural policy while being also easy to analyze; *verification*, in which an off-the-shelf SMT solver verifies the safety of the approximated policy; *shielding*, in which the behaviour of the neural policy is monitored in real-time and the synthesized (approximated) policy is used as a shield to substitute unsafe actions suggested by the neural policy. This differs from our work for two main reasons: first, we work on partially observable environments instead of completely observable environments, and our logical formulas work on beliefs instead of states (i.e., the formulas that we have to use to describe the policy are computationally more expensive and cannot be used in the same way); second, our approach does not require a full description of the behaviour of the policy and can be used to focus the investigation on the most critical properties of a policy. Furthermore, in [55] the logical formulas are used as an input to the verification tool that uses them to verify safety properties, while we use logical formulas directly on the POMCP policy to generate human-readable representations of some of its properties. These logical formulas are then used to detect anomalous actions and perform a shield.

2.4. Explainability

Explainable Artificial Intelligence (XAI) [5] is a rapidly growing research field [56] focusing on human interpretability and understanding of artificial intelligence (AI) systems. In particular, Explainable planning (XAIP) [7,6,57] aims at developing planning tools that come with justifications for the decisions they make. These explanations are particularly important when agents and humans interact [58–60]. Our methodology supports this interaction by using the expert's insight as a basis for the explanation. This leads to compact explanations that can also be used to highlight when the policy and the expert's expectations are different. A particularly interesting class of questions analyzed in XAIP is known as *contrastive*

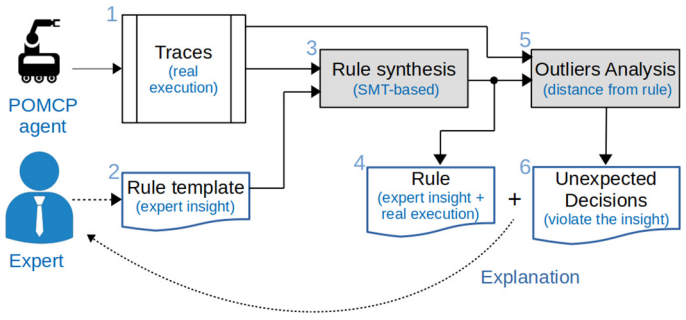


Fig. 1. Overview of the XPOMCP approach.

questions [6], in which a user can ask an agent why it takes a certain decision instead of another one that the expert believes to be better. The system then should answer by motivating the advantage of its choice over the alternative one. However, contrastive questions are difficult to answer in online frameworks because the information required is often not available to the agent. We, therefore, do not use contrastive questions but ground the interaction between humans and planners on logical formulas. These formulas frame the expert's insight and can be used to identify the decisions in contrast with the expert's expectations. While it is possible to compute metrics that capture interesting properties of POMCP's generated policies [61], the knowledge provided by the expert proves to be crucial for proper computing representations. In particular, our approach can be used as an iterative process in which the expert can refine these logical rules to acquire a new understanding of specific properties of the policy by analyzing the execution traces generated by a POMCP agent. A detailed example of iterative refinement of logic rules performed by XPOMCP has been presented in [62].

A methodology that builds explanations based on logical formulas by analyzing a series of events is presented in [63]. The approach is based on temporal logic and decision trees, and it builds sequential explanations (i.e., a relation between an event and past events in the sequence). This is different from our approach because a rule generated by XPOMCP is a compact representation of a policy (i.e., a function that maps beliefs to actions) based on MAX-SMT formulas, not a sequential explanation for events in a time series based on temporal logic. In particular, we build a compact representation of the policy by combining the analysis of multiple execution traces into a single rule to highlight the important aspects of the policy and not a justification for the decisions in a specific trace. Another logic-based methodology is presented [64–67]. It presents an architecture that combines non-monotonic logical reasoning and incomplete knowledge bases to build explanations for the decisions taken by a complex robotic system. This is different from our approach for two main reasons. First, XPOMCP generates rules that present a compact summary of the policy behaviour, not on-demand explanations of actions based on a knowledge base. Second, we base our method on MAX-SMT, not non-monotonic logic. This difference is important because it leads to handling the domains' uncertainty differently. Specifically, we consider uncertainty as a probability distribution over possible domains, while the non-monotonic logic is based on defeasible inferences.

3. Method

In this section, we provide a full description of the proposed method. Moreover, we introduce a formal language that can be used to express templates, and we show how to use information about unexpected decisions returned by our method to generate a shield that identifies the risk involved in selecting an action and blocks risky actions during execution.

3.1. Method overview

The *XPOMCP* framework is summarized in Fig. 1. It leverages the expressiveness of logical formulas to represent specific properties of the investigated policy. As a first step, a logical formula with free variables is defined (see box 2 in Fig. 1) to describe a property of interest of the policy under investigation. This formula, called *rule template*, defines a relationship between some properties of the belief (e.g., the probability of being in a specific state) and an action. Free variables in the formula allow the expert to avoid quantifying the limits of this relationship. These limits are then determined by analyzing a set of observed traces (see box 1 in Fig. 1). For instance, a template saying "Perform this action when the probability of avoiding collisions is at least x", with x free variable, can be transformed into the logical rule saying "Perform this action when the probability of avoiding collisions is at least 0.85". By defining a rule template, the expert provides useful prior knowledge about the structure of the investigated property. Hence, the rule template defines a sort of *question* asked by the expert. The *answer* to this question is provided by the SMT solver (see box 3 in Fig. 1), which computes optimal values for the free variables to allow the formula to explain as many actions as possible in the observed traces.

The rule (see box 4) provides a local representation of the policy function that incorporates the expert's knowledge, and it allows to split trace steps into two classes, namely, those satisfying the rule and those not satisfying it. The approach, therefore, allows identifying unexpected decisions (see box 6) related to actions that violate the logical rule (i.e., that do not verify the expert's assumption). The quantification of the severity of the violation (i.e., the distance between the rule

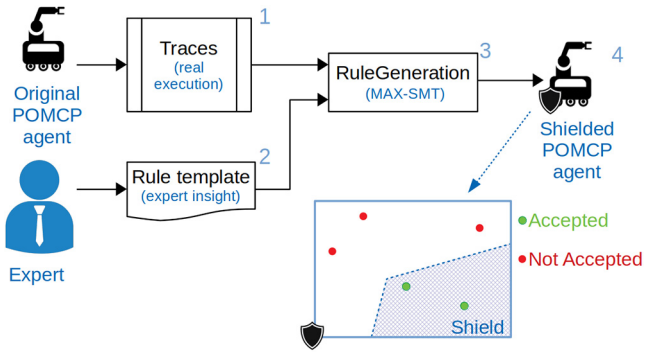
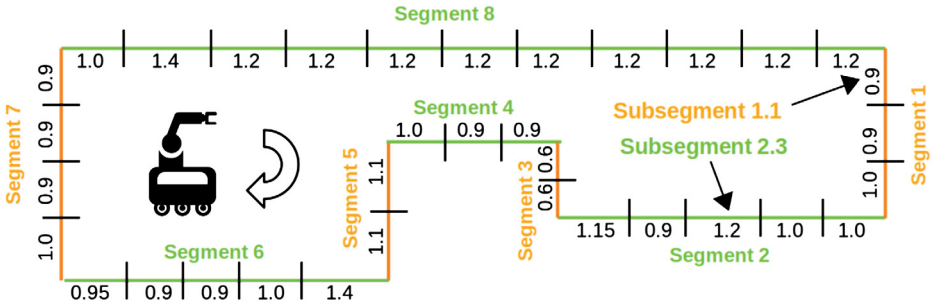


Fig. 2. Shielding.



(a)

f	$p(o=1 \mid f)$
0	0.44
1	0.79
2	0.89

(b)

f	a	$p(c=1 \mid f,a)$
0	0	0.0
0	1	0.0
0	2	0.028
1	0	0.0
1	1	0.056
1	2	0.11
2	0	0.0
2	1	0.14
2	2	0.25

(c)

Fig. 3. Main elements of the POMDP model for the velocity regulation problem. (a) Path map. The map presents the length (in meters) for each subsegment. (b) Occupancy model $p(o \mid f)$: probability of observing a subsegment occupancy given segment difficulty. (c) Collision model $p(c \mid f, a)$: collision probability given segment difficulty and action.

boundary and the violation) also supports the analysis because it provides a straightforward way to explain the violations themselves, which could even be completely unexpected (e.g., due to imprecise expert knowledge or policy errors).

This analysis can be integrated into POMCP as a *shield* to block unexpected and risky decisions proactively. Fig. 2 shows an overview of the shielding mechanism. XPOMCP builds a rule from a rule template and a set of traces (boxes 1, 2, and 3 of Fig. 2), and the shield is based on these results. This shield is then integrated into POMCP (see box 4 in Fig. 2) to filter in real-time the actions that are unexpected in the current belief. Section 3.6 presents the shielding mechanism in detail and Section 3.7 shows how this mechanism has been extended to risk-aware rules.

3.2. Running example: velocity regulation in mobile navigation

We present a problem of velocity regulation in robotic platforms as a case study to show how XPOMCP works. The same problem is also used in Section 4 to evaluate the performance of our method. A robot travels on a pre-specified path divided into eight *segments* which are in turn divided into *subsegments* of different sizes, as shown in Fig. 3. Each segment has a (hidden) difficulty value among *clear* ($f=0$, where f is used to identify the difficulty), *lightly obstructed* ($f=1$) or *heavily obstructed* ($f=2$). All the subsegments in a segment share the same difficulty value. Hence, the hidden state-space has 3^8 states. The robot's goal is to travel on this path as fast as possible while avoiding collisions. In each subsegment, the robot must decide a *speed level* a (i.e., action). We consider three different speed levels, namely 0 (slow), 1 (medium speed), and 2 (fast). The reward received for traversing a subsegment is equal to the length of the subsegment multiplied by $1+a$, where

a is the agent's speed, namely the action that it selects. The higher the speed, the higher the reward, but a higher speed suffers a greater risk of collision (see the collision probability table $p(c = 1 \mid f, a)$ in Fig. 3.c). The real difficulty of each segment is unknown to the robot (i.e., hidden part of the state), but in each subsegment, the robot receives an observation, which is 0 (no obstacles) or 1 (obstacles) with a probability depending on segment difficulty (see Fig. 3.b). The state of the problem contains a hidden variable (i.e., the difficulty of each segment) and three observable variables (current segment, subsegment, and time elapsed since the beginning).

We are interested in a rule describing when the robot travels at maximum speed (i.e., $a = 2$). We expect that the robot should move at that speed only if it is confident enough to be in an easy-to-navigate segment, but this level of confidence varies slightly from segment to segment (due to the length of the segments, the elapsed times, or the relative difficulty of the current segment in comparison to the others). To obtain a compact but informative rule, we want the rule to be a local approximation of the robot's behaviour. Thus we only focus on the current segment without considering the path as a whole when we write this rule. The task of the proposed method is to find the actual bounds on the probability distribution (i.e., belief) that the POMCP algorithm uses to make its decisions and highlight the (unexpected) decisions that do not comply with this representation.

3.3. A language for defining rule templates

To facilitate the usage of the methodology, we introduce a language to write *rule templates*, namely, logical formulas that express high-level expert insight. In particular, the language specifies the elements of a POMCP execution we want to use in our rule templates (i.e., actions, beliefs, and problem-specific information) and how these elements should be combined. The syntax is based on SMT-LIB [10], a standardized language that can be used to write SMT problems. Unlike SMT-LIB, we use a more readable infix notation for the operators. We present the constructs of the language alongside its application to the *velocity regulation* domain. Section 4 presents the usage of this language in three other domains.

3.3.1. Header

To write a useful rule template, we need to specify which elements of the problem we want to describe. The elements of POMCP executions are stored in *traces*. A trace contains one (or more) execution of the POMCP algorithm. Each execution is called a *run* or *episode* and it is a sequence of *steps*. A step corresponds to a belief-action pair where the action is selected by POMCP according to the belief, possibly containing other information (e.g., the position of a robot on a grid). Traces are stored using *eXtensible Event Stream (XES)* [68], a standard format used to collect execution traces of programs. These information are collected in the header of a rule template and are organized as follows:

$$\texttt{actions} = \{id, \ldots, id\}\ type;$$

$$\texttt{belief} = type;$$

$$\texttt{problemInfo} = \{id\ type, \ldots, id\ type\};$$

$$\texttt{runInfo} = \{id\ type, \ldots, id\ type\};$$

$$\texttt{stepInfo} = \{id\ type, \ldots, id\ type\};$$

The *actions* keyword is used to specify the list of actions available to the agent and thus the actions that can be used in the rule templates. All these actions are of the same type (e.g., string or integers). Note that the type is usually specified after the identifier of a variable, as in SMT-LIB. We support most of the type provided by XES. In particular, *bool*, *int*, *real*, and *string* as simple types, and *list* and *map* as aggregate types. Since it is very common to work with probability, we add the *prob* type, which is a real number in the interval $[0, 1]$. The keyword `belief` is used to specify the type used to store the states of the belief (usually encoded as simple integers, used as a unique identifier). The belief is a map that contains the number of particles assigned at each state in the current belief. Finally, *problemInfo*, *runInfo* and *stepInfo* are used to define extra information specific to the domain. With *problemInfo*, it is possible to specify generic information valid for all the runs, for example, the size of a grid or the length of a certain segment in a map. They are expressed as a list of identifiers (a uniquely defined string) with an associate type. Similarly, *runInfo* (*stepInfo*) is used to add information specific to a certain run (step). For example, for the velocity regulation problem we specify the header:

$$\texttt{actions} = \{Speed_L, Speed_M, Speed_F\}\ int;$$

$$\texttt{belief} = int;$$

$$\texttt{problemInfo} = \{segments\ int, subsegments\ list[int]\};$$

$$\texttt{runInfo} = \{\};$$

$$\texttt{stepInfo} = \{seg\ int, subseg\ int\};$$

It contains three actions, namely, $Speed_L$, $Speed_M$ and $Speed_F$ (i.e., slow, medium, and fast speed). In *problemInfo*, the variable *segments* stores the total number of segments and the variable *subsegments* stores the number of subsegments in each segment. Each belief state is stored as an integer that encodes the expected distribution of difficulties among the segments. In particular, the number is computed by raising the expected difficulty of each segment (i.e., 0, 1, or 2) by the id of the segment (i.e., $0, \dots, segments - 1$). We also store the current segment (i.e., *seg*) and subsegment (i.e., *subseg*), that is the visible part of the state, for each step using *stepInfo*. To simplify the usage of these information in our template, a *run* (*step*) variable is always defined, it is used to retrieve information encoded in *runInfo* (*stepInfo*). We also store the number of the *run* and *step* under investigation retrievable as *run.num* and *step.num*.

3.3.2. Variable and function declaration

To express high-level concepts, we use free variables that capture the information stored in the trace. Free variables are declared using:

`declare-var` *id*, ..., *id type*;

It is also possible to declare functions to be used in rule templates:

`define-fun` *id (id type, ..., id type) type* {formula};

the parentheses include the list of the parameters, while last *type* is the type of the function. The *formula* is an SMT expression based on the usage of SMT-LIB. When used in a rule template, these functions are instantiated using the proper values and variables at each step. Functions are useful to encode information compactly. In particular, we introduce a function *p* that takes a state and returns the probability in the belief for that state, useful for working with probabilities and risks (i.e., a number in $[0.0, 1.0]$) instead of the number of particles (e.g., $p(segment_1 hard)$ return 0.1 if there are 100 of the 1000 total particles in states that consider segment one difficult).

For example, to explain the behaviour of the agent in the velocity regulation problem we need to reason on the expected difficulty of the current segment. To express this in a compact way, we introduce the `diff` function which takes a belief `b`, a segment `s`, and a required difficulty value `d` as input and returns the probability that segment `s` has difficulty `d` in the belief `b`.

3.3.3. Rule templates

It is possible to specify one (or more) rule template as:

`declare-rule`

`action` a_1 *rel* $formula_1$;

...

`action` a_n *rel* $formula_n$;

[`where` *requirements*;]

for each *action* statement, we specify the action involved in the rule using a_i. It can be one of the actions specified in the *actions* declaration or a combination of multiple actions grouped using an $\vee$ operator (e.g., to express that two actions have an equivalent effect under certain circumstances). The action and the formula are combined using $rel \in \{ \implies, \impliedby, \iff \}$. With the $\implies$ operator, we specify that an action is performed only when the formula is satisfied. With the $\impliedby$ operator, we specify that we must select that action when the formula is satisfied. Finally, using $\iff$, we specify that both requirements must be satisfied. A $formula_i$ is an SMT formula that combines free variables and information stored in the trace. Our language is based on SMT-LIB. Thus we support all the mathematical operators included in the language for writing these formulas. A copy of the formula is instantiated for each step. The methodology tries to satisfy as many of these formulas as possible, as described in Section 3.4. The (optional) `where` statement can be used to specify a set of hard requirements that the final rule must satisfy, such as the definition of a minimum value for a free variable (e.g., $x_0 \geq 0.9$). These are useful to force desired specifications in the final rule (e.g., to specify that the risk involved in an action must be lower than a fixed value).

For example, to express the idea that the robot in the velocity regulation problem should move at high speed only if it is confident that the current segment is clear we can write the rule template:

`declare-var` x_1, x_2 *prob*;

`declare-rule`

`action` $Speed_F \iff \texttt{diff}(belief, seg, 0) \geq x_1 \vee \texttt{diff}(belief, seg, 2) \leq x_2$

`where` $x_1 \geq 0.9$

Algorithm 1: RuleSynthesis.

Input: a trace generated by POMCP *ex*,
a rule template *r*
Output: an instantiation of *r*

1 $solver \leftarrow$ probability axioms;
2 **foreach** *action rule* r_a *with* $a \in A$ **do**
3 **foreach** *step t in ex* **do**
4 build new dummy literal $l_{a,t}$;
5 $cost \leftarrow cost \cup l_{a,t}$;
6 compute $p_0^t, \ldots, p_n^t$ from *t.particles*;
7 $r_{a,t} \leftarrow$ instantiate rule r_a using $p_0^t, \ldots, p_n^t$;
8 **if** $t.action \neq a$ **then**
9 $r_{a,t} \leftarrow \neg(r_{a,t})$;
10 $solver.add(l_{a,t} \vee r_{a,t})$;
11 $solver$.minimize($cost$);
12 $fitness \leftarrow 1 - distance\_to\_observed\_boundary$;
13 $model \leftarrow solver$.maximize($fitness$);
14 **return** *model*

The first literal of action statement specifies that we select action $Speed_F$ (i.e., high speed) if the probability to be in a segment *seg* with low difficulty (i.e., $\texttt{diff}(belief, seg, 0)$) is greater than a certain threshold x_1. The second literal specifies that we select action $Speed_F$ if the probability of being in a segment with high difficulty (i.e., $\texttt{diff}(belief, seg, 2)$) is less than a certain threshold x_2. Since the two literals are combined with an $\vee$, we select $Speed_F$ if and only if at least one of them is true. The *where* clause containing the expression $x_1 \geq 0.9$ requires x_1 to be at least 0.9 (an information that we expect to be true).

3.4. Rule synthesis

Rule synthesis is the core of our methodology. It is used to generate *rules* from rule templates by instantiating free variables to explain as many as possible of the decisions taken by a POMCP policy in the set of observed traces. The implementation, presented in Algorithm 1, takes as input a log *ex*, generated by POMCP and stored in XES format (as explained in Section 3.3) and a rule template *r*. The output is the rule *r* with all free variables instantiated. The *solver* (we use Z3 [69]) is first initialized and hard constraints are added in line 1 of Algorithm 1 to force all variable of type *prob* in the template to satisfy the probability axioms (i.e., to have value in range $[0, 1]$). Then in the *foreach* loop in lines 2–10 the algorithm maximizes the number of steps that satisfy the rule template *r*. In particular, for each action rule r_a, where a is an action, and for each step t in the trace *ex* the algorithm first generates a literal $l_{a,t}$ (line 4) which is a dummy variable used by MAX-SMT to satisfy clauses that are not satisfiable by a free variable assignment. This literal is then added to the *cost* objective function (line 5) which is a pseudo-boolean function collecting all literals. This function counts the number of fake assignments that correspond to unsatisfied clauses. Afterwards, the belief state probabilities are collected from the particle filter (line 6) and used to instantiate the action rule template r_a (line 7) by substituting their probability variables p_i with observed belief probabilities. This generates a new clause $r_{a,t}$ which represents the constraint for step t. This constraint is considered in its negated form $\neg(r_{a,t})$ if the step action $t.action$ is different from a (line 9) because the clause $r_{a,t}$ should not be true.

The set of logical formulas of the solver is then updated by adding the clause $l_{a,t} \vee r_{a,t}$. In this way, the added clause can be satisfied in two ways, namely, by finding an assignment of the free variables that makes the clause $r_{a,t}$ true (the expected behaviour) or by assigning a true value to the literal $l_{a,t}$ (unexpected behaviour). However, the second kind of assignment has a cost since the dummy variables have been introduced only to allow partial satisfiability of the rules. In line 11, the solver is asked to find an assignment of a free variable that minimizes the cost function, considering the number of dummy variables assigned to true. This minimization is a typical MAX-SMT problem in which an assignment maximizing the number of satisfied clauses is found. Since there can be more than a single assignment of free variables that achieves the MAX-SMT goal, the last step of the synthesis algorithm (lines 12–13) concerns the identification of the assignment, which is closer to the behaviour observed in the trace. This problem is solved by maximizing a fitness function that moves the free variables assignment as close as possible to the numbers observed in the trace without altering the truth assignment of the dummy literals. This problem concerns the optimization of real variables, and the linear arithmetic module solves it.

Notice that MAX-SMT is an NP-hard problem, but in practice, Z3 can solve our instances in a reasonable time (as shown in Section 4) due to the limited number of variables involved and the structure of our instances. Specifically, the variables used in the SMT problem are the free variables specified in the template (a constant number, usually small) and the dummy literals that are linear on the size of the trace because the algorithm builds a clause for each step, and each clause introduces a new dummy literal. Z3 employs numerous heuristics to achieve high performance. In particular, by employing subsumption heuristics, it is possible to remove many redundant steps (i.e., steps generated from similar beliefs) from the problem and thus reduce the number of dummy literals involved significantly.

Algorithm 2: Shielding procedure.

Input: a belief b, a shield Sh, safe action a_{safe}
Output: set of legal actions $\mathcal{L}$

1 $\mathcal{L} \leftarrow \emptyset$;
2 **foreach** *action* $a \in \mathcal{A}$ **do**
3 **if** $a \notin Sh$ **then**
4 $\mathcal{L} \leftarrow \mathcal{L} \cup a$;
5 **else if** $Sh.test\_constraints(b)$ **then**
6 $\mathcal{L} \leftarrow \mathcal{L} \cup a$;
7 **else if** $\exists r \in Sh.Repr : H^2(b, r) < Sh.\tau$ **then**
8 $\mathcal{L} \leftarrow \mathcal{L} \cup a$;
9 **if** $\mathcal{L} = \emptyset$ **then**
10 $\mathcal{L} \leftarrow \{a_{safe}\}$;
11 **return** $\mathcal{L}$;

3.5. Identification of unexpected decisions

A key element of XPOMCP concerns the characterization of steps that fail to satisfy the rule. They can provide useful information for policy interpretation. We define two important classes of exceptions, namely, those related to the approximation made by the logical formula and those actually due to unexpected policy behaviour (e.g., an error in the POMCP algorithm or a decision that cannot be described with only local information). Exceptions in the first class fall quite close to the rule boundary, while exceptions in the second class are usually more distant from the boundary. In the following, we call the second kind of exceptions *unexpected decisions* since their behaviour is unexpected compared to the expert knowledge on the policy. Unexpected decisions are particularly important when the rule template and the policy behaviour are very different. This could happen when the POMCP agent is wrongly implemented or if the expert provides an erroneous template. These cases present, in general, significant and numerous unexpected decisions that highlight the conflict between templates. For example, if we wrongly specify that the robot should move at high speed only when it is confident that the segment is very hard to navigate (instead of very easy), XPOMCP reports many unexpected decisions. The most severe of them is as follows: "the robot decides to move fast even if its confidence in being in an uncluttered segment is 99%", a statement highlighting what is wrong in our template. However, if we use a correct template (i.e., I expect the robot to move fast when the segment is uncluttered) but the POMCP agent is using a wrong policy (e.g., because of parameter issues), then the logic rule generated from the template will detect unexpected decisions. An example of such unexpected decisions could be: "In this observed sample the robot decides to move *fast* when its confidence of being in a cluttered segment is *85%*". This highlights a factual error in the POMCP agent (since we know that moving fast if the segment is cluttered is very risky) that requires further investigation from the designer.

We provide a procedure to identify unexpected decisions. Its input is: a learned rule r, a set of steps (called *steps*) that violate the rule, and a threshold $\tau \in [0, 1]$. The algorithm's output is a set of steps related to unexpected decisions. The procedure first randomly generates N beliefs $\bar{b}_j$, $j = 1, \ldots, N$, that satisfy the rule (e.g., we use $N = 1000$ in our experiments). Then, for each belief b_i in *steps*, a distance measure is computed between b_i and all $\bar{b}_j$, $j = 1, \ldots, N$. The minimum distance h_i is finally computed for each b_i and compared to a threshold τ. If $h_i \geq \tau$ then b_i is considered an outlier because its distance from the rule boundary is high.

Since beliefs are discrete probability distributions, we use a specific distance measure to deal with these elements. Namely, the *discrete Hellinger distance* [70]. This distance is defined as follows:

$$H^2(P, Q) = \frac{1}{\sqrt{2}} \sqrt{\sum_{i=1}^{k} (\sqrt{P_i} - \sqrt{Q_i})^2}$$

where P, Q are probability distributions and k is the discrete number of states in P and Q. An interesting property of H^2 is that it is bounded between 0 and 1, which simplifies thresholding. In Section 4 we discuss how we set this threshold for our experiments.

3.6. Shielded POMCP

It is possible to use the logical rules generated by XPOMCP as a shield, namely a mechanism that forces a POMCP agent to respect the high-level directions provided by the expert through a rule template. We integrate the shield into POMCP to preemptively prune undesired actions considering the current belief. Specifically, we prune the actions before the simulation step. Thus, POMCP only considers valid actions during its search. The shield includes a set of rules trained as explained in Section 3.4 and a set of representative beliefs generated as described in Section 3.5. To shield the actions of the POMCP agent, we use the procedure presented in Algorithm 2. We start with a empty set of *legal actions* $\mathcal{L}$ (line 1).

We add an action a to $\mathcal{L}$ if it satisfies at least one of three possible conditions, namely, *i)* there is no rule for the action in the shield (line 3), *ii)* the current belief b satisfies the constraints defined by the rule for a (line 5, where we use the `test_constraints` function to check the constraints), *iii)* the current belief b does not satisfy the rule, but the distance between b and the boundaries defined by the rule for a are below the threshold τ defined in the shield Sh (line 7). These conditions could result in an empty set of legal actions $\mathcal{L}$ (i.e., if the rules are very strict). In this case, it is important to define a default safe action a_{safe} that is used when no other action is possible. We force POMCP only to consider legal actions in the first step of the simulation. After a legal action is selected, the Monte Carlo Tree Search is performed as usual. Notice that, when the original implementation of POMCP [3] selects a particle in the simulation process, it assumes that the state encoded by the particle is the current state of the system (which for a POMDP is not observable), and thus the belief can only be considered in the first step. With this mechanism, we can ensure that the rule of the shield is respected, but we do not force POMCP to select a specific action. The best action is still decided using the regular POMCP but only among the legal ones.

The computation of legal actions is performed only once for a simulation step since the current belief does not change until a new observation is received from the real environment. Checking that the belief satisfies the constraints (i.e., the `test_constraints` function) has a fixed cost, checking the H^2 of the representative beliefs increases linearly with the number of beliefs. As shown in Section 4, this is a negligible cost, and the reduced number of actions that must be tested (because not all actions are now legal) can also reduce the execution time.

3.7. Risk-aware shielding

The proposed shielding mechanism can handle any logical rule that defines a relationship between beliefs and actions. In this paper, we propose an approach for defining effective rules that exploit this mechanism by directly quantifying the *risk* involved in executing a POMCP policy. We consider the risk as the probability $r = Pr(\bar{s}|b, a)$ that an undesired state $\bar{s} \in S$ is obtained performing an action $a \in A$ from a belief $b \in B$. Thus, the risk depends on the current belief and the chosen action. Undesired states are states that we want to avoid (e.g., a collision or a state where we deplete all our resources). The only way to avoid undesired states in standard POMDPs is by setting high negative rewards in transactions brought to these states. However, in risk-aware applications, it is important to have policies that make explicit the risk and select actions accordingly. POMCP considers rewards to determine the action to perform in a certain belief, but the relationship between the reward and the risk is not explicit and often impossible to evaluate. This could be a problem in safety-critical applications where the risk of following the plan suggested by the policy could become higher than what the system designer expects or can accept.

To overcome this limitation, we provide a method to integrate an explicit computation of the risk into our rules. For example, to quantify the risk of collision in *velocity regulation* we use the information presented in the black-box simulator (and in Fig. 3.c). The risk r of a belief b when the selected action is $Speed_F$ is

$$r = p_0 \cdot 0.028 + p_1 \cdot 0.11 + p_2 \cdot 0.25.$$

The function considers the probability of being in a segment of difficulty 0, 1 or 2 (respectively, p_0, p_1 and p_2), and it multiplies these values to the parameters presented in Fig. 3.c. We can include this information in a rule by using the function

$$\texttt{define-fun risk}(p_0\ prob, p_1\ prob, p_2\ prob)\ real$$
$$p_0 \cdot 0.028 + p_1 \cdot 0.11 + p_2 \cdot 0.25$$

We define rules that limit the risk of actions to acceptable values as *risk-aware rules*. Specifically, this limit must be defined as a hard constraint in the rule formalization (i.e., as an expression in the `where` statement). This ensures that all the decisions taken by the policy must respect this requirement. For example, we can build a risk-aware rule on *velocity regulation* by extending the previously defined rule on the fast moving action $Speed_F$:

$$\texttt{declare-var}\ x_1, x_2\ prob;$$
$$\texttt{declare-rule}$$
$$\texttt{action}\ \ Speed_F \iff p_0 \geq x_1 \vee p_2 \leq x_2$$
$$\texttt{where risk}(x_1, 1 - x_1 - x_2, x_2) < 0.05$$

Notice that multiple combinations of x_1 and x_2 can satisfy the hard requirement to have a risk of collision lower than 5%. However, by including these two free variables in the action rule for $Speed_F$ we ensure that the values are as close as possible to the boundaries of the POMCP policy and the declaration of the risk in the rule allows explicit control of the risk, with an improvement of risk-awareness.

Finally, we define *risk-aware shield* a shield that is computed using risk-aware rules. Notice that properly characterizing the risk involved in a choice of actions makes it possible to shield decisions that have a (negative) impact on future execution steps. This allows us to handle domains that cannot be shielded using only the approach proposed in [13] because the

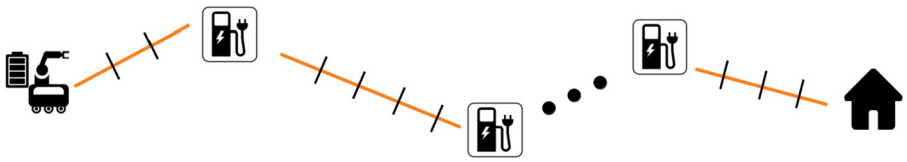


Fig. 4. Visual representation of the *battery* domain.

original shielding mechanism reasons only on the immediate impact of a decision. In particular, in section 4.3.4, we present a domain, i.e., *battery*, where we shield an action that can lead to a (potential) failure several steps ahead (i.e., deploying the battery far away from a recharge station).

4. Results

We test our methodology in four domains. Two of them, *rocksample* and *tiger* [1], are well-known benchmarks in the literature. The other two are *velocity regulation*, used so far as a running example, and *battery*, in which a mobile robot must manage a limited battery to reach its goal. In section 4.3, we use XPOMCP to detect unexpected decisions in the three domains. For *tiger*, we present a comparison between our methodology and a state-of-the-art anomaly detection algorithm called *Isolation Forest* [11] which uses a deterministic optimal exact policy as ground truth. For *rocksample*, *velocity regulation*, and *battery*, it is not feasible to compute optimal exact policies. Thus, we only provide an analysis of the results. However, the effectiveness of their anomaly detection capabilities is empirically evaluated in terms of performance improvement provided by the shield. Specifically, in Section 4.4, we present results for the shielding mechanism in the four domains. These shields incorporate risk-aware requirements. We evaluate the effectiveness of this approach by comparing the performance (cumulative reward) of the original (unshielded) implementation of POMCP with that of the shielded POMCP.

4.1. Domains

The four domains used to evaluate our approach are described in the following.

4.1.1. Tiger

Tiger is a well-known benchmark problem [14] in which an agent has to choose which door to open among two doors, one hiding a treasure and the other hiding a tiger. Finding the treasure yields a reward of $+10$ while finding the tiger a reward of -100. The agent can also listen (by paying a small penalty of -1) to gain new information. Listening is, however, not accurate since there is a 0.15 probability of hearing a roar from the wrong door.

4.1.2. Rocksample

Rocksample is a common POMDP problem in which a robot must sample some (potentially valuable) rocks with the aid of a noisy sensor. The robot moves in a grid (size 11×11 in our experiments) and the rocks (11 in total) are placed in fixed locations known to the agent. These rocks can be valuable or not. Sampling a valuable rock yields a reward of $+10$ while sampling a non-valuable rock yields a reward of -100. The robot can move North, South, East, or West and each movement has a reward of -1 when the robot remains inside the grid. If the robot exits on the North, West, or South border it receives a reward of -100 (it is a scenario that we want to avoid), while it receives a reward of 10 if it exits from the eastern border. The run ends when the agent exits from one of the borders. The agent can check if a rock is valuable using a noisy sensor, which has a precision that decreases the farther the rock is from the robot. Thus, the real value of a rock is partially observable.

4.1.3. Velocity regulation

In this domain, a robot must travel on a predefined path as fast as possible while avoiding collisions. The difficulty of each segment is unknown, and the robot receives (noisy) information on the real difficulty of the segment after each move. A full description of the problem is reported in Section 3.2, where we introduce the problem as a running example.

4.1.4. Battery

In *battery*, a robot must reach a target location while managing a limited battery. The agent follows a path divided into different segments. When the robot crosses a segment, it has an 80% chance of consuming a battery unit. The proper level of the battery is only partially observable, and the robot can use a (noisy) observation to estimate the current values. Specifically, there is a 5% chance of overestimating the level and a 5% chance of underestimating it. The robot knows the positions of segments containing a recharge station. It is possible to fill the battery while paying a fixed cost (a negative reward of -5). If the robot remains without energy in a segment that does not contain a recharge station, it fails, and it receives a negative reward of -100, while if it reaches its goal, it achieves a reward of 100. Our experiments consider a path divided into 35 segments with a variable number of stations. The distance between a station and the next one is in the $[3, 6]$ interval. The robot has a battery of 10 unit. Fig. 4 shows a visual representation of the domain.

4.2. Experimental setting

We implemented the three domains as black-box simulators in the original C++ implementation of POMCP [3]. We extend the implementation to collect *traces* in XES format. The *RuleSynthesis* algorithm (i.e., Algorithm 1) and the procedure for identifying *unexpected decisions* (see Section 3.5) have been developed in Python. The Python binding of Z3 [69] has been used to solve the SMT formulas. The shielding mechanism is implemented in C++ as an extension of the original POMCP implementation. Experiments have been performed on a notebook with Intel Core i7-6700HQ and 16 GB RAM. An implementation of the XPOMCP methodology and the shielding mechanism is available at https://github.com/GiuMaz/XPOMCP.

4.2.1. Error injection

To quantify the capability of the proposed method to identify policy errors and perform shielding, we introduce errors in the parameter tuning of POMCP. In *tiger*, *velocity regulation*, and *battery*, we consider errors in the *reward range* (called c in the following, using the notation of [3]). This parameter must be hand-tuned to achieve high performance in POMCP, and setting it to the wrong value is a frequent mistake that can happen in practice. UCT uses c to balance exploration and exploitation. If this value is lower than the correct one, the algorithm could find a reward that exceeds the maximum expected value leading to a wrong state. Namely, the agent believes to have identified the best possible action, and it stops exploring new actions, even though the selected action is not the best one. This creeping error randomly affects action selection, making POMCP sub-optimal in some situations. This is a hard-to-detect error, thus it is ideal for testing our methodology. The *rocksample* domain is, however, less sensible to errors in parameter c hence, to test our methodology in this domain, we inject the error by modifying another parameter, the *number of simulations* n_p. This parameter defines how many simulations are performed during the MCTS (which corresponds to the number of particles in the particle filter). Using fewer simulations makes POMCP faster, but generated policies are less reliable. Many heuristics used to improve the sample efficiency of POMCP depend on n_p, thus this is an important parameter for testing the effectiveness of our approach.

4.2.2. Exact solution

We use the *incremental pruning algorithm* [1] implemented in [71] to compute an exact policy for *tiger*. This is used as ground truth for evaluating the performance of our method in detecting wrong actions. Unfortunately, we cannot compute the exact policy for the other domains, since their size makes the computation intractable. However, we use these domains to evaluate the applicability of our method to larger problems, and we measure the effectiveness of the generated rules using the shielding mechanism.

4.2.3. Baseline method for detecting unexpected decisions

Isolation forest (IF) [11] is an anomaly detection algorithm that we use as a benchmark for evaluating the performance of our procedure in identifying unexpected decisions. We use it in the *tiger* domain, because it is possible to compute a correct policy, thus each possible anomaly can be properly classified as correct or not using the ground truth. IF is an unsupervised method, and we use it to show that XPOMCP can outperform it while exploiting the rule templates to quantify the benefit of using high level knowledge. IF considers anomalies as rare events and can be applied to a training set containing both nominal and anomalous samples. Hence, it is a good candidate for comparison with XPOMCP. We use the Python implementation of IF provided in *scikit-learn* [72] and consider each step of a trace (i.e., a pair *belief, action*) as a sample (notice that the action is not used as a label). The algorithm uses the *contamination* parameter ρ (i.e., the expected percentage of anomalies in the dataset) to set the threshold used to identify which points are anomalies.

4.3. Detection of unexpected decisions

We focus on the *tiger* problem to test the capabilities of our methodology in detecting unexpected decisions. It is possible to compute a complete policy that works as a baseline for evaluating performance in this domain. We also provide an analysis of the performance of unexpected decision detection for *rocksample*, *velocity regulation* and *battery*. These use cases show the capability of XPOMCP to scale to larger instances.

4.3.1. Unexpected decisions in tiger

A successful policy for *tiger* should listen until enough information is collected about the position of the tiger and then open a door when the agent is reasonably certain to find the treasure behind it (i.e., when the agent considers the risk of being eaten acceptable). From the analysis of the observation model and reward function, it is not immediate to define what "reasonably certain" means. We create a rule template specifying a relationship between the confidence (in the belief) over the treasure position and the related opening action to investigate it. Then we learn the rule parameters from a set of traces performed using POMCP. Finally, by analyzing the trained rule, we understand which is the minimum confidence required by the policy to open a door. A proper value of c is 110 (the reward interval is $[-100, 10]$). For each value of c in $\{110, 85, 65, 40\}$, we generate 50 traces with 1000 runs each, using different seeds for the pseudo-random algorithm in every trace. For each run, we use 2^{15} particles and a maximum of 10 steps per episode. Traces, runs and steps are used as defined in Section 3.3.1. Lower values of c produce a higher number of errors, as shown in Table 1 (see column *% errors*).

Table 1
Comparison between the performance of XPOMCP and that of Isolation Forest in terms of Area Under Curve (AUC) and Average Precision (AP) in tests performed with different reward range (c) and related error rate (*%errors*). Standard deviations are in parenthesis and the best results are highlighted in bold.

c	% errors	AUC_{XPOMCP}	AUC_{IF}	AP_{XPOMCP}	AP_{IF}
110	0.0	–	–	–	–
85	0.0004	**0.993**($\pm$0.041)	0.964($\pm$0.024)	**0.986**($\pm$0.082)	0.057($\pm$0.1076)
65	0.0203	**0.999**($\pm$0.001)	0.992($\pm$0.001)	**0.999**($\pm$0.002)	0.539($\pm$0.0520)
40	0.2374	**0.995**($\pm$0.034)	0.675($\pm$0.020)	**0.987**($\pm$0.084)	0.333($\pm$0.0153)

Rule synthesis To formalize the *tiger* problem within our language, we use the header:

$\texttt{actions} = \{Listen, Open_R, Open_L\}\ int;$

$\texttt{belief} = Bool;$

$\texttt{problemInfo} = \{\};$

$\texttt{runInfo} = \{\};$

$\texttt{stepInfo} = \{\};$

The header presents three actions (i.e., $Listen$, $Open_R$, $Open_L$) and a Boolean belief (i.e., true when we believe that the tiger is behind the left door, false otherwise). No extra information is required in this domain. To formalize the property that the agent has to gather enough confidence in the tiger position before opening a door, we use the following rule template:

$\texttt{declare-var}\ x_1, x_2, x_3, x_4\ prob;$

$\texttt{declare-rule}$

$\quad \texttt{action}\ Listen \iff (p(right) \leq x_1 \wedge p(left) \leq x_2);$

$\quad \texttt{action}\ Open_R \iff p(right) \geq x_3;$

$\quad \texttt{action}\ Open_L \iff p(left) \geq x_4;$

$\texttt{where}\ (x_1 = x_2) \wedge (x_3 = x_4) \wedge (x_3 > 0.9);$

The action rule template for action $Listen$ describes when the agent should listen. Similarly, template action rules for $Open_R$ and $Open_L$ describe when the agent should open the right and left doors. To make the formula more readable, we use $p(left)$ instead of $p(true)$ to refer to the true part of the Boolean belief and, similarly, $p(right)$ instead of $p(false)$. Some hard clauses are also added (in the bottom) to state that *i)* the problem is expected to be symmetric (i.e. the thresholds used to decide when to listen and when to open are the same for both doors, namely $x_1 = x_2$ and $x_3 = x_4$), *ii)* minimum confidence is required to open the door (namely, $x_3 > 0.9$, hence the door should be opened only if the agent is at least 90% sure to find the tiger behind it). This last requirement is particularly important. A safety specification forces the agent never to take a decision that has more than a 10% risk of failure.

Performance evaluation across different thresholds Both XPOMCP and IF use a threshold to identify anomalous points. We use the *Receiver Operating Characteristic (ROC) curve* and the *precision/recall curve* of the two methods to compare the performance across thresholds. The ROC curve considers the relationship between the true positive rate (tpr) and the false positive rate (fpr) at different thresholds. We use the *Area Under Curve* (AUC) as a performance measure. Similarly, the precision/recall curve considers the relationship between precision and recall at different thresholds, and we use the *Average Precision* (AP) as a performance measure. Performance are compared on traces generated using $c \in \{85, 65, 40\}$. We do not evaluate the methods in the case with $c = 110$ because it is error-free (AUC and AP are 0).

In Table 1 we compare the average performance of the two methods. We test XPOMCP with a uniform sampling of 100 thresholds in the interval $[0, 0.5]$. Similarly, we use IF with 100 different values for the contamination parameter ρ uniformly distributed in the interval $[0, 0.5]$. XPOMCP outperforms IF in nearly every instance in both AUC and AP. For both AUC and AP, the difference is high with *c=40*. This is because XPOMCP effectively exploits the information in the template to avoid being influenced by the number of errors. IF performs poorly also with *c=85* because it exhibits a large number of false-positive that leads to very low precision and value of AP. Finally, with *c=65*, the difference between the two methods is smaller. In this dataset, both methods achieve their best performance. This happens because, in this context, most of the errors ($\sim$2% of the dataset) follow the same pattern (i.e., immediately opening the door without any listening action). Thus, it is extremely easy to detect them as anomalies, and nearly all values of τ and ρ achieve maximum performance (i.e., AUC is close to 1). However, in this context, IF generates some false positives, and thus the value AP_{IF} is significantly lower than AP_{XPOMCP}. IF is effective in identifying true errors with $c = 85$, but it still generates many false-positive. Fig. 5 displays two box-plots that show how the AUC and the AP vary for each execution trace. We compute these values as:

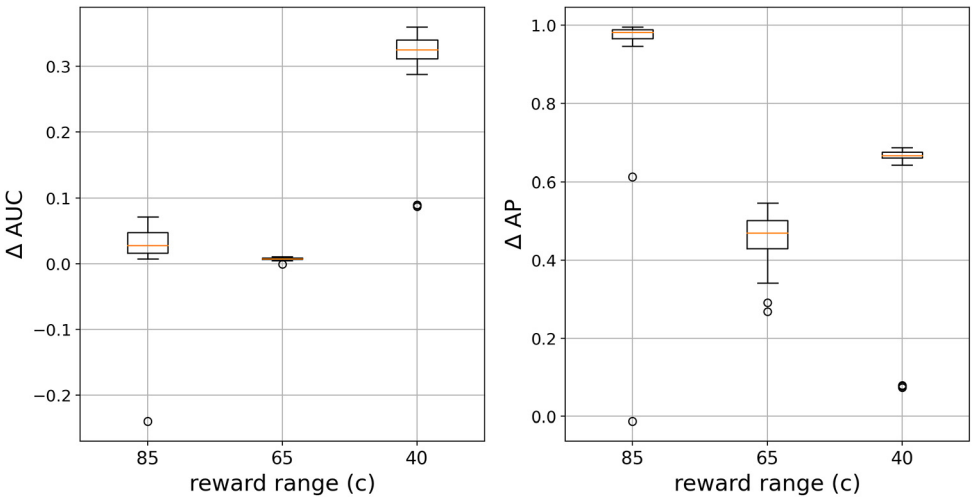


Fig. 5. Box-plots of AUC and AP (considering 100 different parameters) with different values of c.

Table 2
Comparison between the performance of XPOMCP and that of Isolation Forest in terms of F1-score, accuracy and time in tests performed using the best threshold τ across different reward ranges (c) and related error rate (*%errors*). Standard deviations are in parenthesis, and the best results are highlighted in bold.

(a) XPOMCP

c	% errors	τ	F1	Accuracy	time (s)
85	0.0004	0.061	**0.979**($\pm$0.081)	**0.999**($\pm$0.0001)	14.30($\pm$0.50)
65	0.0203	0.064	**0.999**($\pm$0.002)	**0.999**($\pm$0.0001)	14.75($\pm$0.80)
40	0.2374	0.045	**0.980**($\pm$0.072)	**0.987**($\pm$0.049)	12.78($\pm$0.83)

(b) Isolation Forest

c	% errors	ρ	F1	Accuracy	time (s)
85	0.0004	0.01	0.020($\pm$0.033)	0.990($\pm$0.001)	**0.72**($\pm$0.013)
65	0.0203	0.03	0.771($\pm$0.044)	0.988($\pm$0.001)	**0.71**($\pm$0.010)
40	0.2374	0.5	0.437($\pm$0.035)	0.585($\pm$0.026)	**0.64**($\pm$0.037)

$$\Delta AUC = AUC_{XPOMCP} - AUC_{IF}$$
$$\Delta AP = AP_{XPOMCP} - AP_{IF}$$

Since a positive value in the box-plot means that XPOMCP outperforms IF the plot shows that our algorithm is consistently better than IF except for an outlier with $c = 85$.

Performance evaluation with optimal thresholds To provide further details on the performance of XPOMCP, here we show its performance with optimal threshold τ (see Section 3.5). To compute the value of τ, we performed cross-validation by generating rules using 5 traces and testing them on 45 traces. The F1 score for identifying unexpected decisions was computed on the test set using 100 threshold values uniformly distributed in $[0, 0.5]$, and the test with the best F1 score was selected. The results of this test are presented in Table 2.a. Column τ contains values of thresholds used, and columns *accuracy* and *F1* show the related performance values on the test set, and column *time* shows the average elapsed time (in second). We used the same procedure to tune the *contamination* parameter (ρ) of IF (Table 2.b). Fig. 6 compares the average F1 score and accuracy achieved by the two approaches in each test (the value in parenthesis presents the standard deviation). This comparison shows that with optimal parameters, XPOMCP always outperforms IF. Both methods achieve high accuracy due to the high number of non-anomalous samples in the dataset (anomaly and non-anomaly classes are unbalanced), and both methods compute several true negatives. However, the F1 score is very different. IF achieves a low score in this metric because it cannot identify some true positives, and it generates many more false positives than XPOMCP. In general, IF is faster than XPOMCP by an order of magnitude, but the performance of our methodology is acceptable since it takes POMCP an average of 158 seconds to generate a *tiger* trace with 1000 runs, and XPOMCP analyze it in less than 15 seconds.

Analysis of a specific trace To complete our analysis on *tiger*, we show the rule generated by XPOMCP on the analysis of a specific trace generated by POMCP using a wrong value of c, namely, $c = 40$. The rule generated by the MAX-SMT solver from this trace is:

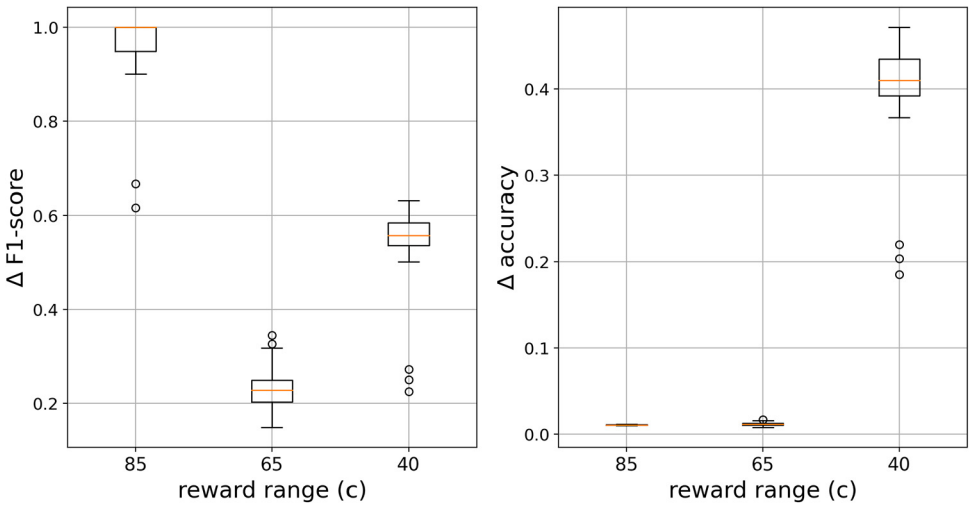


Fig. 6. Box-plots of Δ F1-score and Δ accuracy using the optimal thresholds with different values of c.

$$\begin{aligned}
&\texttt{action}\ Listen \iff (p(right) \leq 0.847 \wedge p(left) \leq 0.847);\\
&\texttt{action}\ Open_R \iff p(right) \geq 0.966;\\
&\texttt{action}\ Open_L \iff p(left) \geq 0.966;
\end{aligned} \tag{1}$$

It is a compact summary of the policy that highlights the important details in a structured way. Notice that this result leverages the information expressed with the hard constraint $x_3 > 0.9$ (i.e., $x_3 = 0.966$ for action $Open_R$). There is a gap between the rule's value for listening (i.e., 0.847) and opening (i.e., 0.966). The trace does not contain any belief in this gap, and XPOMCP cannot build a rule to describe how to act in these beliefs.

Among the total 2659 trace steps, 1601 satisfies the rule, and 1058 does not satisfy it. For all steps not satisfying the rule, we computed their Hellinger distance using the procedure described in Section 3.5. To classify unexpected decisions, we use the optimal threshold of $\tau = 0.045$ (Table 2). This procedure identifies 637 of the 1058 unsatisfiable steps as anomalous (i.e., their H^2 is above threshold τ). In this case, we achieve an F1-score of 1.0 and an accuracy of 1.0 using the proposed approach, and it takes XPOMCP 12.509s to compute this solution. This confirms the ability to detect unexpected decisions even when state-of-the-art anomaly detection methods have degraded performance (i.e., IF reaches an F1-score of 0.59 in this test, but it only takes 0.791s to do so).

4.3.2. Unexpected decisions in rocksample

Rocksample is a challenging problem since its state space grows quickly with the dimension of the board and the number of rocks. For instance, *rocksample(11,11)*, which uses 11 rocks in a 11×11 board has 247,808 states samples [3]. To achieve good performance in such large problems without using too many particles, POMCP uses some ad-hoc heuristics to simplify the computation. In particular, it identifies a subset of interesting actions at each step and increases their starting values in the MCTS. This improves the performance of UCT regarding the exploration-exploitation trade-off since the most promising actions are favoured. However, even with this heuristic *rocksample* requires a significant number of simulations to identify which action is optimal, and the process can fail if not enough simulations are performed. We are interested in applying our methodology in this challenging domain, to capture these errors. We inject errors by changing the number of particles/simulations n_p, and we use XPOMCP to quantify the impact of this parameter. As in *velocity regulation*, it is not possible to compute an optimal policy for *rocksample* as a ground truth. The rules formulated in this section are then used as a base to test the risk-aware shielding mechanism in Section 4.4.

We write a rule template for the riskiest action, namely, rock sampling. We are interested in understanding the confidence level required before selecting the *sample* action (i.e., the action in which the agent collects the rock and gets a related reward). This is a risky decision with a big impact on the final outcome.

To describe the problem, we use the header

$$\texttt{actions} = \{North, East, South, West, sample, check_{1,\dots,11}\}\ int;$$

$$\texttt{belief} = int;$$

$$\texttt{problemInfo} = \{\};$$

$$\texttt{runInfo} = \{Rocks\ list[position]\};$$

$$\texttt{stepInfo} = \{pos\ position\};$$

Table 3
Rule for action *sample* of *rocksample*. For each number of particle n_p the table the *prob* u computed by XPOMCP. Columns *# unsafe sample* and *# unnecessary check* show the two kinds of unexpected decisions for action *sample*.

n_p	u	# unsafe sample	# unnecessary check
2^{18}	0.835	0	1
2^{17}	0.818	0	4
2^{16}	0.827	0	3
2^{15}	0.800	0	7
2^{14}	0.801	15	0
2^{13}	0.800	35	1
2^{12}	0.800	135	0
2^{11}	0.801	237	1

There are four actions for movement, one for sampling, and eleven for checking. The states used in the belief are integers that encode the true value of the eleven rocks (for instance, state 0 corresponds to all valueless rocks, and state 247808 corresponds to all valuable rocks). At each step, we store the (x, y) position of the agent (i.e., the *pos* value, $pos.x$, $pos.y$ can be used to specify the x and y coordinates). Finally, we store the predetermined position of the rocks in the *runInfo* field. To properly describe the behaviour of the robot, it is important to keep track of which rocks were sampled and which ones were not. To do that, we introduce the `collected(r, n)` function, it takes a rock r from *runInfo* and a number of steps n as input and returns *true* if and only if r was sampled at a step lower than n. We can write it as

$$\texttt{define-fun collected}(r\ rock, n\ step)\ bool$$
$$\exists\, n'\ s.t.\ (n'.run = n.run) \wedge (n'.num < n.num) \wedge$$
$$(n'.action = sample) \wedge (n'.x = r.x) \wedge (n'.y = r.y);$$

This function controls whether it exists a previous step in the same run performed a sample action on the same rock.

We compute rules from different traces using different n_p, namely, 2^i particles with $i \in \{11, \ldots, 18\}$. The same number of simulations is performed. We select these parameters because 2^{18} provides very good performance, as shown in [3], and 2^{11} is the bare minimum to represent the possible states of the 11 rocks.

We create a template that explains when the agent considers it worthy to sample a rock:

$$\texttt{declare-var}\ u\ prob;$$
$$\texttt{declare-rule}$$
$$\texttt{action}\ sample \iff \bigvee_{r \in \{rocks\}} (pos = r.pos \wedge \neg \texttt{collected}(r, step) \wedge p(r.valuable) \geq u);$$
$$\texttt{where}\ u \geq 0.8;$$

The above rule specifies that we should use the sample action if and only if we are in the position of a rock (i.e., $pos = r.pos$), the rock was not collected in a previous step (i.e., $\neg\texttt{collected}(r, step)$), and the probability that the rock in this position is valuable is above a certain unknown threshold (i.e., $p(r.valuable) \geq u$). The $\bigvee$ is used to compactly represent an *or* condition over the various rocks. This rule is very important because sampling is the most dangerous action, as it can lead to great rewards if used correctly or significant penalties otherwise. In particular, the parameter u evaluates the risk of sampling the rock under investigation, and it is the parameter that must be kept under control for the purpose of risk-aware shielding. Thus we force $u \geq 0.8$ using the `where` clause.

Table 3 shows the results of our analysis. Column n_p shows the number of particles used in POMCP. Column u reports the value computed for u by XPOMCP. Columns *# unsafe sample* and *# unnecessary check* show the two kinds of unexpected decisions that can happen, namely, the samples made with a confidence below u and the checks made on rocks with a confidence above u. As presented in column u, a higher number of particles leads to use a higher level of confidence before sampling a rock. This improves performance since the agent avoids sampling too many worthless rocks. It is important to note that there are always some steps that do not satisfy the rule, and this happens for two reasons, namely, *i)* the agent samples a rock that is unlikely to be valuable (i.e., *# unsafe sample*), *ii)* the agent checks the value of a rock even if the confidence for the rock to be valuable is above the threshold u (i.e., column *# unnecessary check*). The first kind of unexpected decision is the most important since it can lead to poor performance. As shown in Table 3, with n_p greater than 2^{15} no unsafe sampling is performed.

Table 4
Notable unsatisfiable observed samples in velocity regulation ($c = 90$). Columns: identification (*id*), belief (p_0, p_1, p_2), and Hellinger distance (H^2) of each sample.

id	p_0	p_1	p_2	H^2
1	0.335	0.331	0.334	**0.3526**
2	0.261	0.461	0.278	**0.3090**
3	0.671	0.198	0.131	**0.1717**
4	0.678	0.228	0.094	**0.1389**
5	0.775	0.196	0.029	0.0411
6	0.832	0.127	0.041	0.0347
32	0.853	0.126	0.021	0.0109
33	0.826	0.160	0.014	0.0105

4.3.3. Unexpected decisions in velocity regulation

There is always a (possibly very small) risk of collision when the agent moves fast unless the segment is completely clear in the velocity regulation domain. However, due to the noisy nature of the sensor, there is constantly some uncertainty in the agent's belief. The only strategy that guarantees that the agent never collides is always moving at the slowest pace, a strategy that proves to be sub-optimal. POMCP correctly identifies situations in which the risk of collision is acceptable, leading to good performance in practice. However, it is hard to assess the risks involved in deciding to move fast properly. To evaluate XPOMCP on the *velocity regulation* problem, we analyze one-by-one the decisions marked as unexpected by XPOMCP using a wrong c value (we recall that the exact policy cannot be computed for this problem because the state space is too large). A proper value of c for *velocity regulation* is 103 (i.e., the difference between moving at speed 1 in a short segment and colliding vs. going fast in a long subsegment without collisions, i.e., $[0.6 \cdot 2 - 100, 1.4 \cdot 3]$), but we use $c = 90$ to generate wrong decisions. The template used to describe when the robot must move at high speed is the following:

$$\begin{aligned} &\texttt{declare-var}\ x_1, x_2\ \mathit{prob};\\ &\texttt{declare-rule}\\ &\quad \texttt{action}\ \mathit{Speed}_F \iff p_0 \geq x_1 \vee p_2 \leq x_2\\ &\texttt{where}\ \texttt{risk}(x_1, 1 - x_1 - x_2, x_2) < 0.05 \end{aligned} \tag{2}$$

where x_1, x_2 are free variables and p_0, p_2 are defined using the `diff` function presented in Section 3.3.

We run XPOMCP on the trace used in [12] containing 100 runs. XPOMCP returns the following rule:

$$\texttt{action}\ \mathit{Speed}_F \iff p_0 \geq 0.910 \vee p_2 \leq 0.011$$

It takes 70.05 seconds to analyze the trace. Notice that this rule respects the template, and in the worst-case scenario (p_0 as low as possible and p_2 as high as possible), the risk of collision is 0.033. This rule fails on 33 out of 3500 decisions, but only 4 are marked by XPOMCP as unexpected using threshold $\tau = 0.1$, which we select empirically by analyzing the values of H^2 of unexpected decisions. Table 4 shows some of the most significant steps that do not satisfy the rule (which are not necessarily unexpected decisions) in decreasing order of H^2. Column *id* shows an identification number for the step, columns p_0, p_1, p_2 show the probabilities in the beliefs of unexpected actions, column H^2 shows the Hellinger distance of the steps with unexpected actions, and we write in bold the Hellinger distance values greater than the threshold $\tau = 0.1$. Steps 1 and 2 are unexpected behaviours since POMCP decided to move at high speed even if it had poor information on the difficulty of the segment (p_0, p_1, p_2 are close to a uniform distribution). Steps number 3 and 4 are also unexpected. While they are closer to our rule because p_0 is the dominant value in the belief, they are significantly distant from the boundary of the rule and the decision taken by POMCP. Steps from 5 to 33 have beliefs that only slightly violate the rules of the related actions. These steps are not marked as unexpected due to the approximate nature of the rule.

Finally, Table 5 provides an overview of the results achieved in *velocity regulation* using different values of c. All the rules follow a similar structure (i.e., x_1 is high and x_2 is low). However, x_1, which describes the confidence of being in a clear segment, is lower for lower values of c. Instead, x_2 (i.e., the risk of being in a cluttered segment) becomes higher for lower values of c. All the values in column `risk` are lower than 0.05. However, lower values of c lead to a higher number of risky decisions that violate the rule, in which the robot moves fast with a high risk of collision (column *anomaly*). This sometimes leads to a crash (column *failures*).

4.3.4. Unexpected decisions in battery

In *battery*, the robot must balance the cost of recharging the battery with the risk of depleting it. This risk depends on the distance between the current position and the next station. To achieve good performance, POMCP must reason about the effect of its decisions several steps ahead of the current state because the only positive reward is achieved when the goal is reached. The complexity of selecting the optimal sequence of recharge stations increases exponentially with the total

Table 5
Rule for action $Speed_F$ in *velocity regulation*. For each value of reward range c the table presents the *probs* x_1 and x_2 computed by XPOMCP. Column *risk* shows the risk of collision involved in following the rule. Column *failures* reports the number of failed runs (in a total of 100 runs) and *anomaly* shows the number of anomalies detected by the rule.

c	x_1	x_2	`risk`	failures	anomaly
103	0.911	0.007	0.031	0	2
90	0.910	0.011	0.033	1	4
70	0.898	0.027	0.036	9	51
50	0.828	0.070	0.046	31	171

number of recharge stations. The agent must also consider that the battery level is governed by a non-deterministic function (i.e., there is a probability of 80% of reducing the battery level after each step). Thus, it must adapt its strategy to its current belief.

Rule synthesis To describe the problem, we use the header

$$\texttt{actions} = \{Move, Recharge, Check\}\ int;$$

$$\texttt{belief} = int;$$

$$\texttt{problemInfo} = max\_battery\ int;$$

$$\texttt{runInfo} = \{charging\ list[position], goal\ position\};$$

$$\texttt{stepInfo} = \{pos\ position\};$$

The robot can move, recharge its battery (if its current location is in the *charging* list) and check the current battery level. The belief contains a distribution over the possible battery level, defined as integers in the interval $[0, max\_battery]$ (i.e., $max\_battery = 10$ in our experiments). When the agent uses the *Check* action, it receives an integer in the $[0, max\_battery]$ interval that is a noisy estimation of the current level of the battery. Each step contains the current position of the robot. Thus, the robot knows how distant the next recharge station is from its current location, and it can use this information to decide if it is worth paying the price and recharging the battery.

Notice that the action *Move* can lead the robot to a situation in which the battery is depleted. However, unlike *tiger* and *velocity regulation*, we are not interested in writing a rule to shield this (potentially) dangerous action. Instead, we build a risk-aware rule that describes when the robot should decide to recharge its battery. The outcome of the *Move* action is strongly influenced by the battery level. Thus a rule that properly describes when to use the *Recharge* action prevents dangerous situations during the POMCP execution.

Since the need for a recharge depends on the distance from the next station, we write a different rule for each distance. Since all the rules use the same structure, we only report the rule for a situation in which the next station is at distance 3. The rule is as follows

$$\begin{aligned} &\texttt{declare-var}\ u\ prob;\\ &\texttt{declare-rule}\\ &\quad \texttt{action}\ \ Recharge \iff pos \in recharge \wedge success < u;\\ &\texttt{where}\ success = 0.04 \cdot b_1 + 0.36 \cdot b_2 + \sum_{i=3}^{10} b_i \end{aligned} \tag{3}$$

This specifies that the agent should recharge if and only if its position *pos* is the position of a recharge station, and the probability of successfully reaching the next recharge station (i.e., *success* in the formula) is below a certain threshold u. We compute the chance of success using the domain parameters by considering an 80% chance of reducing the battery by one level after each step. Here, b_i is the probability of having a battery level of i in the belief at a specific step. For b_0 the chance of success is zero (the robot immediately fails if it tries to move), and if its battery level is 3 or more, it is guaranteed to succeed in crossing the next path of length 3. However, if the battery level is 1 or 2, the probability depends on the actual usage of the battery. If at least one segment does not consume the battery, it is possible to cross with only two battery units. This happens with a probability of 36% (it can be directly computed since the probability of reducing the battery level after a step is an independent event). Similarly, if we only have one battery unit, the robot can cross if at least the first two segments do not consume the battery (the chance is, however, only 4%). Notice that if the robot reaches the next station with zero battery, it does not fail since it can immediately use *Recharge*. If the distance is larger than 3, this formula becomes more complex, but it follows the same principle. Notice that we use the same threshold u for each rule to ensure that the risk of failure does not depend on the distance.

Table 6 shows the results of our analysis. Column c shows the different values of the reward range used during the execution. We used 2^{15} particles for the simulation and executed 100 runs in each case. Column u reports the value computed

Table 6
Rule for action *Recharge* in *battery*. For each value of reward range c the table presents the *prob* u computed by XPOMCP. Column *failures* reports the number of failed runs (in a total of 100 runs) and *anomaly* shows the number of anomalies detected by the rule.

c	u	failures	anomaly
60	0.9897	0	6
20	0.9899	1	25
5	0.9897	11	322

Table 7
Comparison between shielded and unshielded POMCP for *tiger*. The table shown the average discounted return and the average execution time. Standard deviations are in parenthesis and the best results are highlighted in bold. Column RI shows the relative increase and column $\#SA$ shows the number of shielded actions.

c	No Shield		Shield			
	return	time (s)	return	RI	time (s)	#SA
110	3.702(±0.623)	0.07(±0.02)	3.702(±0.623)	0.00%	0.07(±0.03)	0
85	3.604(±0.630)	0.07(±0.03)	**3.702(±0.623)**	2.72%	0.06(±0.03)	4
60	3.273(±0.643)	0.06(±0.03)	**3.702(±0.623)**	13.11%	0.06(±0.03)	113
40	−4.173(±1.101)	0.04(±0.01)	**3.702(±0.623)**	188.71%	0.05(±0.02)	647

for u by XPOMCP. Column *# failures* reports the number of failed runs. A run fails when the robot has no battery and cannot recharge it. The results show that an under-tuned value of c leads POMCP to wrongly assess the risk of crossing a segment without a recharge. Column *anomaly* presents the number of anomalies detected by the rule. Notice that in the case of $c = 5$, some anomalies (i.e., 122 of the 322 anomalies) are detected because the agent performs two or more recharges in the same station, a strategy that does not provide any real benefit and leads to suboptimal returns. Finally, notice that the rule converges to a similar value for u in all three instances. Thus XPOMCP can build a good rule (i.e., a rule that can correctly identifies risky actions) even if POMCP uses wrongly set parameters.

4.4. Shielding

We test the performance of the shielding mechanism presented in Section 3.6 in *tiger*, *rocksample*, *velocity regulation*, and *battery*. In our experimental setup, we consider different values of c or n_p, and for each domain and value, we create a trace using the unshielded POMCP. Then, we train a rule to be used as a shield using XPOMCP. Notice that these traces could contain errors, but our method based on MAX-SMT allows to generate logic rules robust to these errors. Finally, we run the POMCP again using the shield and the same parameters. We evaluate the performance of the shield by comparing the average discounted return achieved by the two versions of POMCP (original and shielded). We consider 1000 runs for *tiger* and 100 runs for *velocity regulation* and *battery*. We generate 1000 representative beliefs in all cases, and we use $\tau = 0.10$ as a threshold to identify anomalies.

4.4.1. Shielding for tiger

We base our shield on the same rule presented in Section 4.3. We learn the rule parameters from a POMCP trace, creating a shield from this rule. Since this shield gives a rule for all possible actions, it is important to provide a safe action as described in Section 3.6. For this domain, we use $a_{safe} = Listen$. The correct value of c is 110 because the reward interval is $[-100, 10]$. For each value of c in $\{110, 85, 65, 40\}$, we generate a trace with 1000 runs each, using a fixed seed for the pseudo-random algorithm. In each case, we use 2^{15} particles and a maximum of 10 steps. POMCP with a correct value of c produces the optimal policy (we tested that by comparing the decisions taken by POMCP with an exact policy computed using *incremental pruning* [1]). The results are presented in Table 7. The first column shows the different values of c. The second (third) column shows the average return (time) achieved by the original POMCP and the relative standard deviation. The *Shield* section shows the average return and time achieved by POMCP using a shield (columns four and six). Column *RI* shows the relative increase in performance between the two original and shielded POMCP. Finally, column *#SA* shows how many times the shield alters the decision during the execution.

As shown in Table 7, in the first row of column *#SA*, the shield does not interfere with the correct policy. Lower values of c produce a lower average discounted return, as shown in column *return* of the *No Shield* section. In the *Shield* section we present the relative increase (computed as $RI = \frac{shielded - original}{|original|} \cdot 100$) between the original and the shielded version of the POMCP (see column *RI*). This column shows that the benefit of using a shield is greater when the number of errors increases. For the return column, we report in bold values whose difference from their no-shield counterpart is statistically significant according to a paired t-test with 95% confidence. While there is no difference between the two cases in the first row, the difference is statistically significant in the other three rows.

The average return achieved using the shield is the same in all four cases, and this is also identical to the return achieved by the right policy. This is because in *tiger* we can write a shield that perfectly recreates the behaviour of the correct policy

Table 8
Comparison between shielded and unshielded POMCP for *rocksample*. The table shown the average discounted return and the average execution time. Standard deviations are in parenthesis, and the best results are highlighted in bold. Column *RI* shows the relative increase, and column *#SA* shows the number of shielded actions.

n_p	No Shield		Shield			
	return	time (s)	return	RI	time (s)	#SA
2^{14}	13.01(±0.57)	14.29(±5.0)	13.14(±0.56)	0.99%	17.11(±6.98)	15
2^{13}	12.32(±0.58)	8.05(±2.51)	12.39 (± 0.55)	0.05%	8.68(±2.95)	35
2^{12}	10.21(±0.77)	4.12(±1.37)	**11.98 (± 0.54)**	17.20%	4.64(±1.60)	135
2^{11}	8.83(±0.80)	2.13(±0.56)	**10.25 (± 0.58)**	16.01%	2.43(±0.75)	237

(as shown in Section 4.3), a goal that is difficult to achieve in real-world problems. This is particularly interesting because the shields in the cases of $c \in \{85, 65, 40\}$ are obtained by using traces generated with a POMCP implementation that does make some mistakes. As a consequence, the execution traces contain wrong decisions. However, the expert's insight with the MAX-SMT-based analysis of the traces results in a shield with extremely good performance. As shown in the *time* column, in general, the presence of the shield does not noticeably impact in terms of run-time. The shield generation algorithm takes between 10 and 12 seconds to generate the shield in this case. In the last row, the original POMCP is particularly fast. This happens since the erroneous POMCP opens many doors as fast as possible without listening. A strategy leads to runs that achieve very low average return but ends quickly.

4.4.2. Shielding for rocksample

We test *rocksample* with a risk-aware shield for the sampling action using different number of particles n_p. It blocks the sampling of rocks with low confidence of being valuable in the current belief. While many decisions regarding checks and movement could lead to a higher or lower reward, it is important to notice that *sample* is the action with the highest negative impact. Our rule template prevents sampling with low confidence (i.e., we use the `where` clause to force $u \geq 0.8$). As shown in Section 4.3, the agent performs unsafe sampling only when it uses $n_p = 2^{14}$ or less. Thus we focus our analysis on these cases. We use a trace with 100 runs to train and test the shield. The shield generation takes 23 seconds.

Results are presented in Table 8. The first column shows the number of particles used by POMCP. The second (third) column shows the average return (time) achieved by the original POMCP and the relative standard deviation. The *Shield* section shows the average return and time achieved by POMCP using a shield (columns four and six). Column *RI* shows the relative increase in performance between the two original and shielded POMCP. Finally, column *#SA* shows how many times the shield alters the decision during the execution. *Rocksample* is a harder problem than *tiger* and *velocity regulation*, and the RI achieved is lower in this case. However, the difference is significant when we use 2^{11} or 2^{12} particles. In these cases, the shield provides roughly a 15% increase in performance due to the fact that the agent does not sample rocks for which it has a low confidence. In the cases of 2^{13} and 2^{14} particles, the difference in return is minimal because the unsafe sample performed by the agent are a few (i.e., 15 and 35, respectively). Notice that in these cases, the agent never performs an unsafe action thanks to the risk-aware shield. This is an example in which merging POMCP with logic-based rules representing prior knowledge about the domain allows an improvement for scalability, e.g., using 2^{11} particles and the shield we achieve better average performance (i.e., 10.25) than using 2^{12} particles without the shield (i.e., 10.21) in almost half the time (i.e., 2.44 sec vs 4.12 sec).

4.4.3. Shielding for velocity regulation

We use the rule template expressed in Equation (2) to describe when the robot moves at maximum speed. This is the most dangerous action because there is always a risk of collision involved, as explained in Fig. 3.c. We use a trace with 100 runs to train and test the shield. As usual, rules are generated with traces produced by policies using wrong c and then they are used as shield for the same policies, achieving risk reduction and performance improvement. The shield generation takes approximately 50 seconds to generate velocity regulation shields. Table 9 shows the result of the experiment. The first column shows the different values of c. The second (third) column shows the average return (time) achieved by the original POMCP and the relative standard deviation. The *Shield* section shows the average return and time achieved by POMCP using a shield (columns four and six). Column *RI* shows the relative increase in performance between the two original and shielded POMCP. Finally, column *#SA* shows how many times the shield alters the decision during the execution. As in *tiger*, a lower value of c produces a lower return. In this case, the best c is 103 (the difference between a collision when we move slowly in the shortest segment and a fast movement in the longest segment without a collision). The first row of the table shows that, unlike *tiger*, the usage of a shield can improve the performance even when c is correct. In this case, the shield is activated only 7 times (over the 3500 analyzed steps), yielding a 5.38% increment in the return. This happens because the shield blocks the rare cases in which the POMCP simulations are not enough to assess the risk of moving at high speed properly. This yields acceptable performance in general, but sometimes the simulations are not good enough, and the robot takes a decision that the expert considers too risky.

When c decreases, the shield intervenes more often (see column *#SA*) since the error due to the limited number of simulations is combined with the errors generated by an incorrect value of c. Table 9 also shows that a higher number of activation leads to a bigger relative increase in the performance (column *RI*). The difference is statistically significant in the

Table 9
Comparison between shielded and unshielded POMCP for *velocity regulation*. The table shown the average discounted return and the average execution time. Standard deviations are in parenthesis and the best results are highlighted in bold. Column RI shows the relative increase and column $\#SA$ shows the number of shielded actions.

c	No Shield		Shield			
	return	time (s)	return	RI	time (s)	#SA
103	24.716(±3.497)	10.17(±0.68)	**26.045**(±3.640)	5.38%	10.12(±0.24)	7
90	18.030(±3.794)	10.17(±0.23)	**22.680**(±3.524)	25.79%	10.17(±0.24)	12
70	4.943(±5.260)	10.28(±0.23)	**8.970**(±4.556)	81.46%	10.38(±0.23)	51
50	0.692(±5.051)	10.37(±0.23)	1.638(±4.525)	136.53%	10.44(±0.34)	171

Table 10
Comparison between shielded and unshielded POMCP for *battery*. The table shows the average discounted return and the average execution time. Standard deviations are in parenthesis and the best results are highlighted in bold. Column RI shows the relative increase and column $\#SA$ shows the number of shielded actions.

c	No Shield		Shield			
	return	time (s)	return	RI	time (s)	#SA
60	46.79(±0.32)	0.92(±0.30)	48.82(±0.26)	4.33%	0.92(±0.30)	6
20	34.49(±1.36)	1.55(±1.55)	**43.04**(±0.59)	24.78%	1.13(±0.72)	22
5	30.29(±1.88)	1.28(±2.01)	**42.52**(±0.36)	40.37%	1.04(±0.20)	155

case of $c \in \{103, 90, 70\}$, and shows that the introduction of the shield improves the performance up to the 81%, even in cases in which the shield is trained using traces generated by a POMCP process that makes some mistakes. In the case of $c = 50$, the return increases a lot (i.e., $RI = 136, 53\%$), but the difference is not statistically significant because the standard deviation is very high. The shield intervenes 171 times by blocking risky high-speed moves, but unlike *tiger*, in which we use a rule for every possible action, here POMCP made many wrong decisions when it moves at low or medium speed (for example, by moving slowly when the path is clear, possibly because the low value of c leads to an overestimation of the expected return of moving slowly). As in *tiger*, the usage of the shield does not significantly increase the time required to perform the simulations.

4.4.4. Shielding for battery

We test *battery* with a shield for the recharge action based on the rule template expressed in Equation (3). Thus, the shield forces the robot to recharge its battery when the success chance is below an acceptable threshold. We use a trace with 100 runs to train and test the shield. The shield generation takes 3.2 seconds.

Results are presented in Table 10. The table is structured as in Table 9. *Battery* is a challenging domain for POMCP because an optimal recharge strategy must consider many future scenarios. The results show that our shield can drastically improve the performance, with an increment of up to 40.37%, by forcing POMCP to recharge the battery when it is too dangerous to move. Notice that, in this case the correct value is $c = 60$ and the performance achieved with $c = 20$ and $c = 5$ are inferior to the performance achieved with $c = 60$. The wrong c parameter made POMCP charge the battery in a sub-optimal way (e.g., by overcharging the battery when a charge was not necessary). This leads to lower performance. However, the shielded execution can still achieve high performance by avoiding battery depletion.

5. Conclusions and future work

In this work, we present a methodology that combines high-level indications provided by a human expert with an automatic procedure that analyzes execution traces to synthesize key properties of a policy in the form of logical rules. We exploit such rules offline by detecting unexpected decisions in the traces and online by shielding actions related to unexpected decisions of the policy. We show that our methodology can exploit high-level knowledge to outperform a state-of-the-art anomaly detection algorithm. We also show that the shielding mechanism improves the performance of POMCP policies in four application domains. This is achieved by exploiting the high-level knowledge provided by human experts who design the structure of the logical rules, which are then instantiated by a MAX-SMT-based algorithm. Since MAX-SMT allows us to encode requirements as hard and soft clauses, our approach is robust to errors and can work on traces generated from suboptimal or wrong policies. Finally, by formulating the policy as a logical rule, we show that it is possible to combine rules and safety requirements to generate risk-aware shields that prevent risky actions in POMCP. This work paves the way toward several interesting research directions.

The fact that experts must write rule templates is both strength and a limitation. It is a strength because it allows human beings to query the policy and discover its strategy. On the other hand, it is a limitation because sometimes rule templates could be complex or unknown even by experts. The formulas that enforce risk awareness use these templates to capture relevant policy elements. Thus, templates play an important role in controlling the system's behaviour. This approach is prone to bias and cannot be applied in domains in which the structure of the problem is not known a priori. A recent work [73], however, presents a methodology based on inductive logic programming (ILP) that directly produces rules (and potentially

rule templates) by analyzing the execution traces. This approach can be used to overcome these limitations. However, it currently requires a significantly larger dataset to produce reliable results. Thus, user-defined templates remain important tools. Furthermore, the approach proposed in [73] cannot be trivially integrated with safety and risk-aware requirements. We aim to extend our methodology to support automatic template generation without losing risk-aware requirements.

Another future research direction consists in developing an active approach to rule synthesis. Currently, the methodology requires a previously generated dataset of execution traces. In practice, many of these traces present redundant information, and this passive approach could be impractical for extremely large domains. We plan to develop an active approach that exploits the information encoded into the Monte Carlo Tree Search to quickly identify significant traces (i.e., traces that provides information that was not previously available). A first result in this direction is presented in [74]. However, this approach cannot be used to assess the risk of selecting an action. Thus, we are working to overcome these limitations.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

This research has been partially funded by project “Dipartimenti di Eccellenza 2018-2022”, Italian Ministry of Education, Universities and Research.

References

[1] A.R. Cassandra, M.L. Littman, N.L. Zhang, Incremental pruning: a simple, fast, exact method for partially observable Markov decision processes, in: D. Geiger, P.P. Shenoy (Eds.), UAI '97: Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence, Brown University, Providence, Rhode Island, USA, August 1-3, 1997, Morgan Kaufmann, 1997, pp. 54–61.
[2] C.H. Papadimitriou, J.N. Tsitsiklis, The complexity of Markov decision processes, Math. Oper. Res. 12 (3) (1987) 441–450.
[3] D. Silver, J. Veness, Monte-Carlo planning in large POMDPs, in: J.D. Lafferty, C.K.I. Williams, J. Shawe-Taylor, R.S. Zemel, A. Culotta (Eds.), Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010, Proceedings of a Meeting Held 6-9 December 2010, Vancouver, British Columbia, Canada, Curran Associates, Inc., 2010, pp. 2164–2172.
[4] L. Kocsis, C. Szepesvári, Bandit based Monte-Carlo planning, in: J. Fürnkranz, T. Scheffer, M. Spiliopoulou (Eds.), Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings, in: Lecture Notes in Computer Science, vol. 4212, Springer, 2006, pp. 282–293.
[5] D. Gunning, D.W. Aha, DARPA's explainable artificial intelligence (XAI) program, AI Mag. 40 (2) (2019) 44–58.
[6] M. Fox, D. Long, D. Magazzeni, Explainable planning, CoRR, arXiv:1709.10256, 2017.
[7] M. Cashmore, A. Collins, B. Krarup, S. Krivic, D. Magazzeni, D.E. Smith, Towards explainable AI planning as a service, CoRR, arXiv:1908.05059, 2019.
[8] N. Jansen, B. Könighofer, S. Junges, A. Serban, R. Bloem, Safe reinforcement learning using probabilistic shields (invited paper), in: I. Konnov, L. Kovács (Eds.), 31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference), in: LIPIcs, vol. 171, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 3:1–3:16.
[9] C.W. Barrett, C. Tinelli, Satisfiability modulo theories, in: E.M. Clarke, T.A. Henzinger, H. Veith, R. Bloem (Eds.), Handbook of Model Checking, Springer, 2018, pp. 305–343.
[10] C. Barrett, P. Fontaine, C. Tinelli, The satisfiability modulo theories library (SMT-LIB), www.SMT-LIB.org, 2016.
[11] F.T. Liu, K.M. Ting, Z. Zhou, Isolation forest, in: Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy, IEEE Computer Society, 2008, pp. 413–422.
[12] G. Mazzi, A. Castellini, A. Farinelli, Identification of unexpected decisions in partially observable Monte Carlo planning: a rule-based approach, in: F. Dignum, A. Lomuscio, U. Endriss, A. Nowé (Eds.), AAMAS '21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021, ACM, 2021, pp. 889–897.
[13] G. Mazzi, A. Castellini, A. Farinelli, Rule-based shielding for partially observable Monte-Carlo planning, in: S. Biundo, M. Do, R. Goldman, M. Katz, Q. Yang, H.H. Zhuo (Eds.), Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (Virtual), August 2-13, 2021, AAAI Press, 2021, pp. 243–251.
[14] L.P. Kaelbling, M.L. Littman, A.R. Cassandra, Planning and acting in partially observable stochastic domains, Artif. Intell. 101 (1–2) (1998) 99–134.
[15] R. Coulom, Efficient selectivity and backup operators in Monte-Carlo tree search, in: H.J. van den Herik, P. Ciancarini, H.H.L.M. Donkers (Eds.), Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers, in: Lecture Notes in Computer Science, vol. 4630, Springer, 2006, pp. 72–83.
[16] S. Katt, F.A. Oliehoek, C. Amato, Learning in POMDPs with Monte Carlo tree search, in: D. Precup, Y.W. Teh (Eds.), Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6–11 August 2017, in: Proceedings of Machine Learning Research, vol. 70, PMLR, 2017, pp. 1819–1827.
[17] C. Amato, F.A. Oliehoek, Scalable planning and learning for multiagent POMDPs, in: B. Bonet, S. Koenig (Eds.), Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA, AAAI Press, 2015, pp. 1995–2002.
[18] J. Lee, G. Kim, P. Poupart, K. Kim, Monte-Carlo tree search for constrained POMDPs, in: S. Bengio, H.M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (Eds.), Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada, 2018, pp. 7934–7943.
[19] A. Castellini, G. Chalkiadakis, A. Farinelli, Influence of state-variable constraints on partially observable Monte Carlo planning, in: S. Kraus (Ed.), Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019, ijcai.org, 2019, pp. 5540–5546.

[20] M. Lauri, R. Ritala, Planning for robotic exploration based on forward simulation, Robot. Auton. Syst. 83 (2016) 15–31.
[21] Y. Wang, F. Giuliari, R. Berra, A. Castellini, A.D. Bue, A. Farinelli, M. Cristani, F. Setti, POMP: pomcp-based online motion planning for active visual search in indoor environments, in: 31st British Machine Vision Conference 2020, BMVC 2020, Virtual Event, UK, September 7–10, 2020, BMVA Press, 2020.
[22] F. Giuliari, A. Castellini, R. Berra, A.D. Bue, A. Farinelli, M. Cristani, F. Setti, Y. Wang, Pomp++: pomcp-based active visual search in unknown indoor environments, in: 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE Press, 2021, pp. 1523–1530.
[23] A. Goldhoorn, A. Garrell, R. Alquézar, A. Sanfeliu, Continuous real time POMCP to find-and-follow people by a humanoid service robot, in: 14th IEEE-RAS International Conference on Humanoid Robots, Humanoids 2014, Madrid, Spain, November 18-20, 2014, IEEE, 2014, pp. 741–747.
[24] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T.P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis, Mastering the game of Go with deep neural networks and tree search, Nature 529 (7587) (2016) 484–489.
[25] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, D. Hassabis, A general reinforcement learning algorithm that masters chess, shogi, and go through self-play, Science 362 (6419) (2018) 1140–1144.
[26] S.A. Seshia, D. Sadigh, Towards verified artificial intelligence, CoRR, arXiv:1606.08514, 2016.
[27] R. McAllister, Y. Gal, A. Kendall, M. van der Wilk, A. Shah, R. Cipolla, A. Weller, Concrete problems for autonomous vehicle safety: advantages of Bayesian deep learning, in: C. Sierra (Ed.), Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, ijcai.org, 2017, pp. 4745–4753.
[28] P.S. Thomas, B.C. da Silva, A.G. Barto, S. Giguere, Y. Brun, E. Brunskill, Preventing undesirable behavior of intelligent machines, Science 366 (6468) (2019) 999–1004.
[29] N. Bjørner, A. Phan, L. Fleckenstein, νZ - an optimizing SMT solver, in: C. Baier, C. Tinelli (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, in: Lecture Notes in Computer Science, vol. 9035, Springer, 2015, pp. 194–199.
[30] R.R. Zakrzewski, Verification of a trained neural network accuracy, in: IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No. 01CH37222), vol. 3, 2001, pp. 1657–1662.
[31] C. Cheng, G. Nührenberg, H. Ruess, Verification of binarized neural networks, CoRR, arXiv:1710.03107, 2017.
[32] X. Huang, M. Kwiatkowska, S. Wang, M. Wu, Safety verification of deep neural networks, in: R. Majumdar, V. Kuncak (Eds.), Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I, in: Lecture Notes in Computer Science, vol. 10426, Springer, 2017, pp. 3–29.
[33] G. Katz, C.W. Barrett, D.L. Dill, K. Julian, M.J. Kochenderfer, Reluplex: an efficient SMT solver for verifying deep neural networks, in: R. Majumdar, V. Kuncak (Eds.), Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I, in: Lecture Notes in Computer Science, vol. 10426, Springer, 2017, pp. 97–117.
[34] N. Narodytska, S.P. Kasiviswanathan, L. Ryzhyk, M. Sagiv, T. Walsh, Verifying properties of binarized deep neural networks, in: S.A. McIlraith, K.Q. Weinberger (Eds.), Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th Innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, AAAI Press, 2018, pp. 6615–6624.
[35] R. Bunel, I. Turkaslan, P.H.S. Torr, P. Kohli, P.K. Mudigonda, A unified view of piecewise linear neural network verification, in: S. Bengio, H.M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (Eds.), Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada, 2018, pp. 4795–4804.
[36] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, M.T. Vechev, AI2: safety and robustness certification of neural networks with abstract interpretation, in: 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA, IEEE Computer Society, 2018, pp. 3–18.
[37] K. Jia, M. Rinard, Efficient exact verification of binarized neural networks, in: H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, H. Lin (Eds.), Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, Virtual, 2020.
[38] G. Norman, D. Parker, X. Zou, Verification and control of partially observable probabilistic systems, Real-Time Syst. 53 (3) (2017) 354–402.
[39] Y. Wang, S. Chaudhuri, L.E. Kavraki, Bounded policy synthesis for pomdps with safe-reachability objectives, in: E. André, S. Koenig, M. Dastani, G. Sukthankar (Eds.), Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018, International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 2018, pp. 238–246.
[40] O. Bastani, Y. Pu, A. Solar-Lezama, Verifiable reinforcement learning via policy extraction, in: S. Bengio, H.M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (Eds.), Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada, 2018, pp. 2499–2509.
[41] A. Verma, V. Murali, R. Singh, P. Kohli, S. Chaudhuri, Programmatically interpretable reinforcement learning, in: J.G. Dy, A. Krause (Eds.), Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018, in: Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 5052–5061.
[42] M. Cashmore, D. Magazzeni, P. Zehtabi, Planning for hybrid systems via satisfiability modulo theories, J. Artif. Intell. Res. 67 (2020) 235–283.
[43] S. Junges, N. Jansen, C. Dehnert, U. Topcu, J. Katoen, Safety-constrained reinforcement learning for MDPs, in: M. Chechik, J. Raskin (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, the Netherlands, April 2-8, 2016, Proceedings, in: Lecture Notes in Computer Science, vol. 9636, Springer, 2016, pp. 130–146.
[44] N. Fulton, A. Platzer, Safe reinforcement learning via formal methods: toward safe control through proof and learning, in: S.A. McIlraith, K.Q. Weinberger (Eds.), Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th Innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, AAAI Press, 2018, pp. 6485–6492.
[45] M. Hasanbeig, A. Abate, D. Kroening, Cautious reinforcement learning with logical constraints, in: A.E.F. Seghrouchni, G. Sukthankar, B. An, N. Yorke-Smith (Eds.), Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020, International Foundation for Autonomous Agents and Multiagent Systems, 2020, pp. 483–491.
[46] R.S. Sutton, A.G. Barto, Reinforcement Learning - an Introduction, Adaptive Computation and Machine Learning, MIT Press, 1998.
[47] S. Junges, N. Jansen, S.A. Seshia, Enforcing almost-sure reachability in POMDPs, in: A. Silva, K.R.M. Leino (Eds.), Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II, in: Lecture Notes in Computer Science, vol. 12760, Springer, 2021, pp. 602–625.
[48] P. Santana, S. Thiébaux, B.C. Williams, RAO*: an algorithm for chance-constrained POMDP's, in: AAAI, AAAI Press, 2016, pp. 3308–3314.
[49] M. Klauck, M. Steinmetz, J. Hoffmann, H. Hermanns, Bridging the gap between probabilistic model checking and probabilistic planning: survey, compilations, and empirical comparison, J. Artif. Intell. Res. 68 (2020) 247–310.
[50] A.A.R. Newaz, S. Chaudhuri, L.E. Kavraki, Monte-Carlo policy synthesis in POMDPs with quantitative and qualitative objectives, in: A. Bicchi, H. Kress-Gazit, S. Hutchinson (Eds.), Robotics: Science and Systems XV, University of Freiburg, Freiburg im Breisgau, Germany, June 22–26, 2019, 2019.

[51] J. Krumm, E. Horvitz, Risk-aware planning: methods and case study for safer driving routes, in: AAAI, AAAI Press, 2017, pp. 4708–4714.
[52] R. Bloem, B. Könighofer, R. Könighofer, C. Wang, Shield synthesis: runtime enforcement for reactive systems, in: C. Baier, C. Tinelli (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, in: Lecture Notes in Computer Science, vol. 9035, Springer, 2015, pp. 533–548.
[53] B. Knighofer, M. Alshiekh, R. Bloem, L.R. Humphrey, R. Könighofer, U. Topcu, C. Wang, Shield synthesis, Form. Methods Syst. Des. 51 (2) (2017) 332–361.
[54] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, U. Topcu, Safe reinforcement learning via shielding, in: S.A. McIlraith, K.Q. Weinberger (Eds.), Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), AAAI Press, 2018, pp. 2669–2678.
[55] H. Zhu, Z. Xiong, S. Magill, S. Jagannathan, An inductive synthesis framework for verifiable reinforcement learning, in: K.S. McKinley, K. Fisher (Eds.), Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019, ACM, 2019, pp. 686–701.
[56] N. Burkart, M.F. Huber, A survey on the explainability of supervised machine learning, J. Artif. Intell. Res. 70 (2021) 245–317.
[57] S. Anjomshoae, A. Najjar, D. Calvaresi, K. Främling, Explainable agents and robots: results from a systematic literature review, in: E. Elkind, M. Veloso, N. Agmon, M.E. Taylor (Eds.), Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13-17, 2019, International Foundation for Autonomous Agents and Multiagent Systems, 2019, pp. 1078–1088.
[58] T. Chakraborti, S. Sreedharan, Y. Zhang, S. Kambhampati, Plan explanations as model reconciliation: moving beyond explanation as soliloquy, in: C. Sierra (Ed.), Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, ijcai.org, 2017, pp. 156–163.
[59] Y. Zhang, S. Sreedharan, A. Kulkarni, T. Chakraborti, H.H. Zhuo, S. Kambhampati, Plan explicability and predictability for robot task planning, in: 2017 IEEE International Conference on Robotics and Automation, ICRA 2017, Singapore, Singapore, May 29 - June 3, 2017, IEEE, 2017, pp. 1313–1320.
[60] P. Langley, B. Meadows, M. Sridharan, D. Choi, Explainable agency for intelligent autonomous systems, in: S.P. Singh, S. Markovitch (Eds.), Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA, AAAI Press, 2017, pp. 4762–4764.
[61] A. Castellini, E. Marchesini, G. Mazzi, A. Farinelli, Explaining the influence of prior knowledge on POMCP policies, in: N. Bassiliades, G. Chalkiadakis, D. de Jonge (Eds.), Multi-Agent Systems and Agreement Technologies - 17th European Conference, EUMAS 2020, and 7th International Conference, AT 2020, Thessaloniki, Greece, September 14-15, 2020, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 12520, Springer, 2020, pp. 261–276.
[62] G. Mazzi, A. Castellini, A. Farinelli, Policy interpretation for partially observable Monte-Carlo planning: a rule-based approach, in: Proceedings of the 7th Italian Workshop on Artificial Intelligence and Robotics (AIRO 2020@AI*IA2020), in: CEUR Workshop Proceedings, vol. 2806, CEUR-WS.org, 2020, pp. 44–48.
[63] A.A.B. da Costa, P. Dasgupta, Learning temporal causal sequence relationships from real-time time-series, J. Artif. Intell. Res. 70 (2021) 205–243.
[64] M. Sridharan, M. Gelfond, S. Zhang, J.L. Wyatt, REBA: a refinement-based architecture for knowledge representation and reasoning in robotics, J. Artif. Intell. Res. 65 (2019) 87–180.
[65] T. Mota, M. Sridharan, Answer me this: constructing disambiguation queries for explanation generation in robotics, in: IEEE International Conference on Development and Learning, ICDL 2021, Beijing, China, August 23-26, 2021, IEEE, 2021, pp. 1–8.
[66] T. Mota, M. Sridharan, A. Leonardis, Integrated commonsense reasoning and deep learning for transparent decision making in robotics, SN Comput. Sci. 2 (4) (2021) 242.
[67] M. Sridharan, T. Mota, Commonsense reasoning to guide deep learning for scene understanding (extended abstract), in: C. Bessiere (Ed.), Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, ijcai.org, 2020, pp. 4760–4764.
[68] G. Acampora, A. Vitiello, B.N.D. Stefano, W.M.P. van der Aalst, C.W. Günther, E. Verbeek, IEEE 1849: the XES standard: the second IEEE standard sponsored by IEEE computational intelligence society [society briefs], IEEE Comput. Intell. Mag. 12 (2) (2017) 4–8.
[69] L.M. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: C.R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, in: Lecture Notes in Computer Science, vol. 4963, Springer, 2008, pp. 337–340.
[70] E. Hellinger, Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen, J. Reine Angew. Math. 136 (1909) 210–271.
[71] E. Bargiacchi, D.M. Roijers, A. Nowé, AI-toolbox: a C++ library for reinforcement learning and planning (with Python bindings), J. Mach. Learn. Res. 21 (2020) 102, 1–12.
[72] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: machine learning in Python, J. Mach. Learn. Res. 12 (2011) 2825–2830.
[73] G. Mazzi, D. Meli, A. Castellini, A. Farinelli, Learning logic specifications for soft policy guidance in POMCP, in: Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems, AAMAS '23, London, United Kingdom, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2023, pp. 373–381.
[74] G. Mazzi, A. Castellini, A. Farinelli, Active generation of logical rules for POMCP shielding, in: AAMAS '22: 21st International Conference on Autonomous Agents and Multiagent Systems, International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2022, pp. 1696–1698.