Machine Learning (2022) 111:3797–3838
https://doi.org/10.1007/s10994-021-06102-7

Lifted model checking for relational MDPs

Wen-Chi Yang[1] · **Jean-François Raskin**[2] · **Luc De Raedt**[3]

Received: 19 May 2020 / Revised: 30 September 2021 / Accepted: 14 October 2021 /
Published online: 23 June 2022

Abstract
Probabilistic model checking has been developed for verifying systems that have stochastic and nondeterministic behavior. Given a probabilistic system, a probabilistic model checker takes a property and checks whether or not the property holds in that system. For this reason, probabilistic model checking provide rigorous guarantees. So far, however, probabilistic model checking has focused on propositional models where a state is represented by a symbol. On the other hand, it is commonly required to make relational abstractions in planning and reinforcement learning. Various frameworks handle relational domains, for instance, STRIPS planning and relational Markov Decision Processes. Using propositional model checking in relational settings requires one to ground the model, which leads to the well known state explosion problem and intractability. We present pCTL-REBEL, a lifted model checking approach for verifying pCTL properties of relational MDPs. It extends REBEL, a relational model-based reinforcement learning technique, toward relational pCTL model checking. PCTL-REBEL is lifted, which means that rather than grounding, the model exploits symmetries to reason about a group of objects as a whole at the relational level. Theoretically, we show that pCTL model checking is decidable for relational MDPs that have a possibly infinite domain, provided that the states have a bounded size. Practically, we contribute algorithms and an implementation of lifted relational model checking, and we show that the lifted approach improves the scalability of the model checking approach.

Keywords Model checking · Probabilistic computation tree logic (pCTL) · First-order logic · Lifted inference · Relational MDPs

1 Introduction

Probabilistic model checking aims at deciding whether a stochastic model satisfies a given probabilistic property (Baier and Katoen 2008; Forejt et al. 2011). By doing so, it provides rigorous guarantees about the model. Markov Decision Processes (MDPs) and Probabilistic Computational Tree Logic (pCTL) (Forejt et al. 2011; Kwiatkowska et al. 2011;

Editors: Nikos Katzouris, Alexander Artikis, Artur d'Avila Garcez, Ute Schmid, Jay Pujara.

✉ Wen-Chi Yang
wenchi.yang@kuleuven.be

Extended author information available on the last page of the article

Dehnert et al. 2017) are standard formalisms for specifying the model and the properties, respectively. MDPs are commonly used for modeling sequential decision making problems where the actions have stochastic effects. PCTL is a temporal logic that expresses model properties over time and allows for probabilistic quantification. A property can be *the machine gives a warning before shutting down with a probability higher than 0.95*, or *the probability is higher than 0.9 that the emergency power supply, after giving a warning, continues to function for at least 10 more minutes*.

It is common in planning and reinforcement learning to make abstraction of the domain elements in order to compactly define models and speed up the computation. However, model checking methods most often operate on explicit-state MDPs (Baier and Katoen 2008) thus do not allow for such abstractions. This is undesirable since the number of states increases exponentially with the domain size, making it infeasible to explicitly traverse the state space (Slaney and Thiébaux 2001). In such large domains, it is impractical to apply model checking techniques as they lead to a state explosion (Otterlo 2004). For instance, the well known blocks world has 501 states for 5 blocks but over 58 million states for only 10 blocks (Slaney and Thiébaux 2001). In this paper, we aim at mitigating such state explosions in probabilistic model checking by making relational abstractions and using lifted inference.

Lifting is the key to scalablility in relational domains (Kersting 2012; Van den Broeck et al. 2011; de Salvo Braz et al. 2007). It is also central to statistical relational AI (StarAI) (De Raedt et al. 2016). Lifting implies reasoning about a group of objects as a whole at the first-order level, and exploiting the shared relational structures and symmetries in the model. This is done by making abstraction of irrelevant details of the objects. As an illustration, an object's full identity (e.g. a block's ID number) can be left out as long as it satisfies the property description (e.g. blue). There has been a significant interest in such relational representations in reinforcement learning and planning. For instance, Džeroski et al. (2001) introduced *Relational Markov Decision Processes (RMDP)*, a first-order generalization of MDPs that succinctly formulates relational models by implicitly defining states in terms of objects and relations (Džeroski et al. 2001; Otterlo 2004). RMDPs have been used in reinforcement learning and planning to compute first-order policies without explicitly constructing the underlying state space (Džeroski et al. 2001; Kersting et al. 2004; Wang et al. 2008; Kersting and De Raedt 2004; Driessens and Džeroski 2004; Boutilier et al. 2001; Sanner and Boutilier 2009; Yoon et al. 2012). One especially interesting example is *REBEL* (Kersting et al. 2004), the RElational BELlman operator, which we will extend in this paper. REBEL is a model-based reinforcement learning technique for constructing an optimal policy in a given RMDP. It is also a lifted inference technique that alleviates state explosions.

Motivated by the success of temporal logics and MDPs in probabilistic model checking and in planning, we investigate whether it is possible to lift these approaches to RMDPs. We show that the answer is positive by introducing *pCTL-REBEL*, a new framework that augments REBEL (Kersting et al. 2004) with pCTL. More specifically, pCTL-REBEL is a relational model checking approach that checks relational pCTL formulae in RMDPs. In addition to mitigating state explosions, it is important to take one step further to investigate lifted probabilistic model checking for infinite models. Although model checking for infinite models is generally undecidable (Gabbay 2003), a rich body of research has discussed the *state-boundedness* assumption that yields decidable verification of infinite systems (Bagheri Hariri et al. 2013; Belardinelli et al. 2011, 2012; De Giacomo et al. 2012; Calvanese et al. 2018). These studies almost exclusively focus on non-probabilistic actions. Nevertheless, they provide great insight into relational model checking with pCTL

properties. We extend the work of Belardinelli et al. (2012) to the probabilistic setting to obtain the decidability of the model checking problem for a subclass of infinite MDPs.

The key contribution of this paper is twofold. First, we introduce a lifted model checking algorithm, pCTL-REBEL, to mitigate the state explosion problem of checking relational MDPs. PCTL-REBEL is fully automated and provides a complete, lifted solution for relational model checking. In order to adapt to the model checking framework, pCTL-REBEL introduces an alternative interpretation of REBEL (Kersting et al. 2004) in which a state value corresponds to the *probability* that a given formula is satisfied by executions starting from that state. Second, while model checking is generally undecidable for infinite MDPs, we provide decidability results for a class of infinite MDPs under the state-boundedness condition. In particular, we prove that a finite relational abstraction exists for RMDPs that have an infinitely large domain, and that checking the relational abstraction is equivalent to checking the infinite MDP. This means that strong guarantees for such infinite MDPs can be provided via lifted model checking, as implemented in pCTL-REBEL.

This paper is structured as follows. Section 2 provides an overview on basic notions of relational MDPs. Section 3 reviews basic notions of model checking and introduces relational pCTL. Section 3.3 defines the problem statement of this paper, that is, probabilistic model checking for relational MDPs. Section 4 defines a relational Bellman update operator (a generalization of REBEL (Kersting et al. 2004)) for relational value iteration. Based on Sect. 4, Sect. 5 introduces the main algorithm, pCTL-REBEL, for relational model checking. Section 6 provides theoretic results about obtaining decidability for a subclass of infinite MDPs. Section 7 reports on experimental evaluation. Section 8 discusses related work and Sect. 9 highlights the link between relational model checking and safe reinforcement learning. Finally, Sect. 10 concludes the work.

2 Relational notions and relational MDPs

This section defines the basic notions of relational MDPs. These notions will be used in the remainder of this paper with the *blocks world* running example. We closely follow the notions of the standard first-order logic (Nienhuys-Cheng and Wolf 1997) and relational MDPs (Kersting et al. 2004).

2.1 Relational logic

Relational logic generalizes propositional logic with *variables* such that a variable represents a set of constants. This section provides an overview of relational logic, following the notions of Nienhuys-Cheng and Wolf (1997).

An *alphabet* is a tuple $\Sigma = \langle R, D \rangle$ where R is a finite set of relation symbols and D is a possibly infinite set of constants. Each relation symbol $\mathtt{p} \in R$ has an arity $m \geq 0$. An *atom* $\mathtt{p}(\mathtt{t}_1, ..., \mathtt{t}_m)$ is a relation symbol $\mathtt{p}$ followed by an m-tuple of terms $\mathtt{t}_i$. A *term* is a variable $\mathtt{W}$ or a constant $\mathtt{c}$. A variable (resp. constant) is expressed by a string that starts with an upper (resp. lower) case letter. A conjunction is a set of atoms, which is implicitly assumed to be *existentially quantified*. A definite clause $H \leftarrow B$ consists of an atom H and a conjunction B, stating that *H is true if B is true*. Given an expression $\mathtt{E}$, *vars*($\mathtt{E}$) (resp. *consts*($\mathtt{E}$), *terms*($\mathtt{E}$)) denotes the set of all variables (resp. constants, terms) in $\mathtt{E}$. An expression is called *ground* if it contains no variables. We shall make the **unique name assumption**, that states all constants are unequal, that is, $\mathtt{c}_1 \neq \mathtt{c}_2$ holds for

different constants $\mathtt{c}_1$ and $\mathtt{c}_2$. A substitution θ is a set of bindings $\{\mathtt{W}_1/\mathtt{t}_1, ..., \mathtt{W}_n/\mathtt{t}_n\}$ that assigns terms $\mathtt{t}_i$ to variables $\mathtt{W}_i$. A grounding substitution θ assigns constants to variables in an expression E such that $\mathtt{E}\theta$ contains no variables.

We shall use the *Object Identity subsumption* framework (OI-subsumption) (Ferilli et al. 2002), which means that any two terms in an expression are unequal, and the pairwise inequalities should be added. For instance, under OI-subsumption, the conjunction $\{\mathtt{cl(b)}, \mathtt{on(X,Y)}\}$ denotes the expression $\{\mathtt{cl(b)}, \mathtt{on(X,Y)}, \mathtt{X} \neq \mathtt{Y}, \mathtt{X} \neq \mathtt{b}, \mathtt{Y} \neq \mathtt{b}\}$. For ease of writing, when the context is clear, we shall not write these inequalities explicitly. A conjunction A is *OI-subsumed* by conjunction B, denoted by $\mathtt{A} \preceq_\theta \mathtt{B}$, if there exists a substitution θ such that $\mathtt{B}\theta \subseteq \mathtt{A}$. Only substitutions that satisfy the inequality constraints are allowed.

A *unifier* θ of two conjunctions A and B under OI-subsumption is a substitution such that $\mathtt{A}\theta = \mathtt{B}\theta$. For example, the conjunctions $\{\mathtt{cl(X)}, \mathtt{on(y,Z)}\}$ and $\{\mathtt{cl(x)}, \mathtt{on(Y,Z)}\}$ have a unifier $\{\mathtt{X/x}, \mathtt{Y/y}\}$. A *maximally general specialization (mgs)* of two conjunctions A and B under OI-subsumption is a conjunction that is OI-subsumed by A and B, and is not OI-subsumed by any other specializations. The mgs operation is not always unique under OI-subsumption. For example, the conjunctions $\{\mathtt{cl(X)}\}$ and $\{\mathtt{on(Y,Z)}\}$ have maximally general specializations $\{\mathtt{cl(X)}, \mathtt{on(X,Y)}\}$ and $\{\mathtt{cl(X)}, \mathtt{on(Y,Z)}\}$ that do not OI-subsume one another.

The *Herbrand base* of an alphabet $\Sigma = \langle R, D\rangle$, denoted by HB^Σ, is the set of all ground atoms that can be constructed from Σ. A *Herbrand interpretation* s is a subset of HB^Σ where all atoms in s are *true* and all others are *false*. The set of all Herbrand interpretations determined by Σ is denoted by S^Σ. We shall write S instead of S^Σ when the context is clear. When the domain D is infinite, the set of all Herbrand interpretations S^Σ must be infinite.

2.2 Relational MDP

Relational MDPs (RMDPs) generalize explicit-state MDPs in a twofold manner. First, RMDPs have structured states. More specifically, an RMDP state is represented by a conjunction of ground atoms whereas an explicit state is represented by a single constant. Second, RMDPs allows variables in the state description. In consequence, a set of RMDP states can be represented by one single *abstract state*, which enables reasoning about a set of states as a whole. In this paper, an RMDP is a variant of the standard RMDP (Kersting et al. 2004; Boutilier et al. 2001) that varies by allowing the domain to be infinite. This section formally defines RMDPs, following the notions of Kersting et al. (2004).

An RMDP is a tuple $K = \langle \Sigma, \Delta\rangle$ where the alphabet $\Sigma = \langle R, D\rangle$ contains a set of relations and a domain, and Δ is a finite set of abstract transitions. The alphabet Σ determines the state space. A state $s \in S^\Sigma$ is a Herbrand interpretation. An *abstract state* s' is then a conjunction of atoms, representing a set of states, denoted by $s'\Theta = \{s \in S^\Sigma | s \preceq_\theta s'\}$.

Example 1 Consider a blocks world with the alphabet $\Sigma = \langle R, D\rangle$ where the relations are $R = \{\mathtt{cl/1}, \mathtt{on/2}\}$ and the domain is $D = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, the abstract state $s = \{\mathtt{cl(A)}, \mathtt{cl(C)}, \mathtt{on(A,B)}\}$ represents the following six ground states.

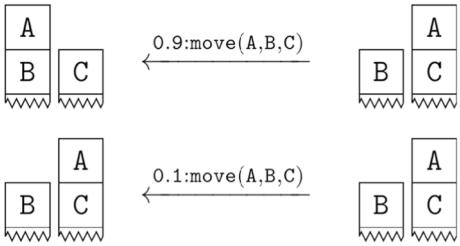

Fig. 1 The abstract transition `move(A, B, C)` moves block `A` to block `B` from block `C` with probability 0.9. It fails to move the block with probability 0.1

$$s\Theta = \{\{\texttt{cl(a),cl(c),on(a,b)}\}, \{\texttt{cl(a),cl(b),on(a,c)}\},$$
$$\{\texttt{cl(b),cl(c),on(b,a)}\}, \{\texttt{cl(b),cl(a),on(b,c)}\},$$
$$\{\texttt{cl(c),cl(b),on(c,a)}\}, \{\texttt{cl(c),cl(a),on(c,b)}\}\}$$

An *abstract action* $\alpha \notin R$ is an atom for an action relation. An *abstract transition* $\delta \in \Delta$ (based on an abstract action α) is a finite set of probabilistic transition rules $\delta = \{H_1 \xleftarrow{p_1:\alpha} B, ..., H_n \xleftarrow{p_n:\alpha} B\}$ where B (resp. H_i) is an abstract state, representing the precondition (resp. postcondition), and $p_i \in [0, 1]$ is the transition probability. These transition rules δ denote a proper probability distribution over H_i, that is, $\sum_{i=1}^{n} p_i = 1$. To ensure that all abstract transitions rely only on information in the current state, we assume that all variables in H_i are also in B, that is, $vars(H_i) \subseteq vars(B)$.

Example 2 A blocks world is defined by an RMDP $K = \langle \Sigma, \Delta \rangle$ where $\Sigma = \langle R, D \rangle$, $R = \{\texttt{cl/1}, \texttt{on/2}\}$ and Δ contains the following abstract transition `move(A, B, C)`.

$$\delta_{\texttt{move}} \begin{cases} \texttt{cl(A),cl(C),on(A,B)} \xleftarrow{0.9:\texttt{move(A,B,C)}} \texttt{cl(A),cl(B),on(A,C)} \\ \texttt{cl(A),cl(B),on(A,C)} \xleftarrow{0.1:\texttt{move(A,B,C)}} \texttt{cl(A),cl(B),on(A,C)} \end{cases}$$

The abstract action `move(A, B, C)` expresses *moving block* `A` *to block* `B` *from block* `C`. The action succeeds with probability 0.9 and fails with probability 0.1. When the action fails, the state stays the same. A graphical illustration is in Fig. 1.

2.3 Grounding an RMDP

The semantics of an RMDP is defined at the ground level such that any RMDP (including the infinite ones) implicitly defines an underlying ground MDP. This section formally defines the construction of the underlying ground MDP. In the end of this section, we will briefly discuss the decidability issue of infinite RMDPs.

Given an RMDP $K = \langle \Sigma, \Delta \rangle$, the underlying ground MDP is a tuple $M = \langle S, A, T \rangle$ where S is a set of ground states, A is a set of ground actions and $T : S \times A \times S \to [0, 1]$ is a ground transition function. Every *ground state* $s \in S$ is a Herbrand interpretation of Σ. A ground state s has a set of available *ground actions*, denoted by $A(s) \subseteq A$. These ground actions $A(s)$ are defined by grounding the abstract actions. Formally,

$$A(s) := \{\alpha\theta | H_i \xleftarrow{p_i:\alpha} B \in \Delta, s \preceq_\theta B\}$$

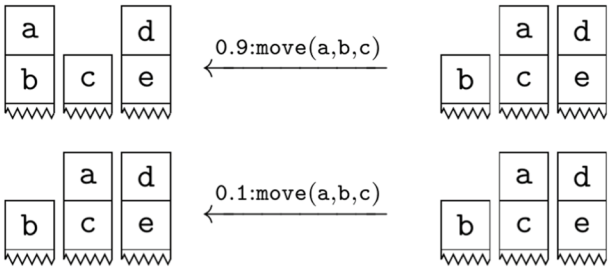

Fig. 2 The ground action move(a, b, c) moves block a to block b from block c with probability 0.9. The action fails with probability 0.1

Similarly, given a ground state s and a ground action $\alpha\theta \in A(s)$, the set of *ground transitions* $T(s, \alpha\theta)$ are defined by grounding the abstract transitions. Formally,

$$T(s, \alpha\theta) := \{h_i \xleftarrow{p_i:\alpha\theta} s | H_i \xleftarrow{p_i:\alpha} B \in \Delta, s \preceq_\theta B, h_i = (s \backslash B\theta) \cup H_i\theta\}$$

After taking action $\alpha\theta$ in state s, the transition probability distribution over all ground states $s' \in S$ is then

$$T(s, \alpha\theta)(s') := p_i, \text{ where } s' = (s \backslash B\theta) \cup H_i\theta$$

Since Δ is a proper abstract transition function, T must be a proper probability distribution, i.e. $\sum_{s' \in S} T(s, \alpha\theta)(s') = 1$.

Example 3 Consider a blocks world defined by an RMDP $K = \langle \Sigma, \Delta \rangle$ where $\Sigma = \langle R, D \rangle$, $R = \{\texttt{cl/1}, \texttt{on/2}\}$, $D = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}\}$ and $\Delta = \{\texttt{move(A, B, C)}\}$ (see Fig. 1). The RMDP K defines the underlying MDP $M = \langle S, A, T \rangle$. One of the resulting ground transitions is as follows. Let a ground state be $s = \{\texttt{cl(a)}, \texttt{cl(b)}, \texttt{cl(d)}, \texttt{on(a, c)}, \texttt{on(d, e)}\} \in S$, and let an action be $\texttt{move(a, b, c)} \in A(s)$, the resulting next state must be $s' = \{\texttt{cl(a)}, \texttt{cl(c)}, \texttt{cl(d)}, \texttt{on(a, b)}, \texttt{on(d, e)}\} \in S$. By taking the $\texttt{move(a, b, c)}$ action in state s, the probability of reaching s' is 0.9, and the probability of staying in s is 0.1, as illustrated in Fig. 2.

In general, for an RMDP that has an infinite domain, the underlying ground MDP has an infinitely large state space and action space. Hence, it is infeasible to explicitly traverse the state space. Moreover, such RMDPs have an unbounded branching behavior such that a state has infinitely many available actions, leading to infinitely many other states. For example, one can move a clear block to any of the (infinitely many) other clear blocks. Therefore, the model checking problem for infinite RMDPs is generally undecidable. To obtain decidability, we will later identify a special class of infinite RMDPs such that the branching behavior is bounded. More details are in Sect. 6.

3 Model checking for relational MDPs

This section defines the problem statement of this paper, namely the *model checking problem of relational MDPs*. More specifically, Sect. 3.1 reviews the fundamentals of model checking, Sect. 3.2 defines the relational pCTL language that will be used to specify properties throughout this paper, and Sect. 3.3 defines the main problem statement. Later, Sects. 4 and 5 will provide a solution for the problem in Sect. 3.3.

3.1 Probabilistic reachability

This section defines the *probabilistic reachability property*, the most fundamental property in model checking. The probabilistic reachability refers to the maximum probability of reaching a set of goal states from a given initial state. This property is required to define the relational pCTL language in Sect. 3.2. The content of this section is standard in model checking (Baier and Katoen 2008; Kwiatkowska et al. 2011) and is included to make this paper self-contained.

We first define *paths* in an MDP and their probability assignment. Given an MDP $M = \langle S, A, T \rangle$, a path of length n is denoted by $\rho_n = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} s_n$ where $s_i \in S$, $a_i \in A(s_i)$ and $T(s_i, a_i)(s_{i+1}) > 0$. Similarly, an infinite path is denoted by $\rho = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$. The set of all finite (resp. infinite) paths is denoted by $FPath_M$ (resp. $IPath_M$), and the set of all finite (resp. infinite) paths starting from state s is denoted by $FPath_{M,s}$ (resp. $IPath_{M,s}$). We denote all paths starting from s in M as $Path_{M,s}$=$FPath_{M,s} \cup IPath_{M,s}$. The i-th state of a path ρ is denoted by $\rho(i)$. The last state of a path ρ is denoted by $last(\rho)$. To project a set of paths $Path_{M,s}$ to a probability space, it is required to remove nondeterministic actions by a policy. A policy $\pi : FPath_M \times A \to [0, 1]$ takes a finite path ρ_n and specifies a probability distribution over all available actions $A(last(\rho_n))$. We write $\pi(\rho_n)$ to denote all possible actions selected by the policy, and we write $\pi(\rho_n, a)$ to denote the probability of action a being selected.

Definition 1 [cf. Forejt et al. (2011)] Given an MDP M and a policy π, the probability $P_M^\pi(\rho_n)$ of a finite path $\rho_n = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} s_n \in FPath_{M,s_1}$ is inductively defined by

$$\begin{aligned} P_M^\pi(\rho_1) &= 1 \\ P_M^\pi(\rho_k) &= P_M^\pi(\rho_{k-1}) \sum_{a \in \pi(\rho_{k-1})} \pi(\rho_{k-1}, a) T(s_{k-1}, a)(s_k) \end{aligned} \tag{1}$$

where ρ_k denotes ρ_n's prefix of length k. Let $C_{\rho_n} \in IPath_{M,s_1}$ be the set of all infinite paths that have a prefix ρ_n (also known as the basic cylinder), the probability $P_M^\pi(C_{\rho_n})$ is then defined as the probability of ρ_n, i.e. $P_M^\pi(C_{\rho_n}) = P_M^\pi(\rho_n)$.

After defining the probability for an MDP path, we can now formally define probabilistic reachability.

Definition 2 [cf. Forejt et al. (2011)] Given an MDP $M = \langle S, A, T \rangle$, an initial state $s \in S$ and a set of goal states $G \subseteq S$, the set of all paths from s to G, denoted by $Path_{M,s}(G)$, is formally defined as follows.

$$Path_{M,s}(G) := \{\rho \in Path_{M,s} | \exists i \in \mathbb{N}, \rho(i) \in G\}$$

By following an optimal policy π, one can generate a path from s to G with a maximum probability. Any other policies in the policy space Π does not achieve a probability larger than the optimal policy π does. The maximum reachability is defined as follows.

$$P_M^{max}(Path_{M,s}(G)) := \sup_{\pi \in \Pi} P_M^\pi(Path_{M,s}(G)) \tag{2}$$

In general, $P_M^{max}(\cdot)$ denotes the maximum reachability in M.

It is known that a deterministic, stationary ϵ-optimal policy suffices to achieve the maximum probabilistic reachability in a finite MDP (Baier and Katoen 2008). This means the policy will simplify to a function $\pi : S \to A$ that maps a path's last state to a single action. For an infinite RMDP that has a finite abstraction (cf. Sect. 6), we assume there exists an optimal deterministic, stationary policy for the maximum probabilistic reachability.

3.2 Relational pCTL

This section introduces *Relational Probabilistic Computational Tree Logic* (relational pCTL), a temporal logic that describes system behavior over time and allows for probabilistic quantification. We will use relational pCTL to specify properties of an RMDP. Relational pCTL is a variant of the standard pCTL [cf. Forejt et al. (2011), Baier and Katoen (2008)] that varies by allowing variables in atoms and only allowing negations in front of atoms.

The syntax of the relational pCTL is as follows. A property is always specified by a state formula ϕ.

$$\begin{aligned} &\textit{state formula } \phi \ ::= \texttt{true} \mid l \mid \neg l \mid \phi \wedge \phi \mid \texttt{P}_{\bowtie \texttt{p}}[\psi] \\ &\textit{path formula } \psi \ ::= \texttt{X}\ \phi \mid \phi\ \texttt{U}^{\leq \texttt{k}}\ \phi \mid \phi\ \texttt{U}\ \phi \end{aligned}$$

where l is an atom (that can contain variables), $\texttt{p}$ is a probability such that $0 \leq \texttt{p} \leq 1$, $\texttt{k} \in \mathbb{N}$ is a step bound and $\bowtie \in \{\leq, <, \geq, >\}$. Here, relational pCTL generalizes the standard pCTL by letting l be a relational atom instead of a constant.

The semantics of the relational pCTL resembles the standard pCTL (Baier and Katoen 2008). A state either satisfies or violates a state formula ϕ, resulting in a boolean evaluation for each state. The $\texttt{X}$ operator stands for *next*, and the $\texttt{U}$ stands for *until*. A path formula $\texttt{X}\phi$ is satisfied if ϕ is satisfied in the next state; $\phi_1 \texttt{U}^{\leq \texttt{k}} \phi_2$ is satisfied if ϕ_2 is satisfied within $\texttt{k}$ steps and ϕ_1 holds before then; $\phi_1 \texttt{U} \phi_2$ is satisfied if ϕ_2 is eventually satisfied and ϕ_1 holds before then.

The semantics of the relational pCTL is defined at the ground level. Given an RMDP $K = \langle \Sigma, \Delta \rangle$ that defines a ground MDP $M = \langle S, A, T \rangle$, we say a ground state $s \in S$ satisfies a state formula ϕ, denoted by $s \vDash \phi$, if and only if there exists a grounding substitution θ for all free variables in ϕ such that s satisfies ϕ under θ, i.e. $s \vDash \phi \Leftrightarrow \exists \theta . s \vDash^{\theta} \phi$. All substitutions must respect OI-subsumption, i.e. any two terms $\texttt{t}_1, \texttt{t}_2$ in a conjunction must be unequal. Formally, the pCTL satisfiability relation $\vDash^{\theta}$ is inductively defined as follows.

$$\begin{aligned} &s \vDash^{\theta} \texttt{true} && \\ &s \vDash^{\theta} l && \Leftrightarrow s \preceq_{\theta} l \\ &s \vDash^{\theta} \neg l && \Leftrightarrow s \npreceq_{\theta} l \\ &s \vDash^{\theta} \phi_1 \wedge \phi_2 && \Leftrightarrow s \vDash^{\theta} \phi_1 \wedge s \vDash^{\theta} \phi_2 \\ &s \vDash^{\theta} \phi_1 \vee \phi_2 && \Leftrightarrow s \vDash^{\theta} \phi_1 \vee s \vDash^{\theta} \phi_2 \\ &s \vDash^{\theta} \texttt{P}_{\bowtie \texttt{p}}[\psi] && \Leftrightarrow P_M^{max}(\{\rho \in P_{M,s} | \rho \vDash \psi\}) \bowtie \texttt{p} \end{aligned}$$

where

$$\begin{aligned}
&\rho \vDash \mathtt{X}\ \phi && \Leftrightarrow \rho(2) \vDash \phi \\
&\rho \vDash \phi_1\ \mathtt{U}^{\leq k}\ \phi_2 && \Leftrightarrow \exists i{\leq}k+1.[\rho(i) \vDash \phi_2 \wedge \forall j{<}i.\rho(j) \vDash \phi_1] \\
&\rho \vDash \phi_1\ \mathtt{U}\ \phi_2 && \Leftrightarrow \exists i{\in}\mathbb{N}.[\rho(i) \vDash \phi_2 \wedge \forall j{<}i.\rho(j) \vDash \phi_1]
\end{aligned}$$

With the aforementioned operators, additional operators can be defined as follows where F stands for *eventually*.

$$\mathtt{false} \equiv \neg\mathtt{true} \quad \mathtt{F}^{\leq k}\ \phi \equiv \mathtt{true}\ \mathtt{U}^{\leq k}\ \phi \quad \mathtt{F}\ \phi \equiv \mathtt{true}\ \mathtt{U}\ \phi$$

Example 4 Consider a formula $\phi = \neg\mathtt{cl(A)}$ that states "there exists an unclear block A in the state". The satisfiability relation $s \vDash \phi$ can be rewritten as follows.

$$\begin{aligned}
& s \vDash \neg\mathtt{cl(A)} \\
\Leftrightarrow\ & \exists\theta \cdot s \vDash^{\theta} \neg\mathtt{cl(A)} \\
\Leftrightarrow\ & \exists\theta \cdot s \not\vDash_{\theta} \mathtt{cl(A)}
\end{aligned}$$

For the ground state $s_1 = \{\mathtt{cl(a)}\}$, the above evaluates to *false* as no θ exists for the satisfiability relation. For another ground state $s_2 = \{\mathtt{cl(a)}, \mathtt{on(a,b)}\}$, the above evaluates to *true* with $\theta = \{\mathtt{A/b}\}$.

Example 5 Consider a formula $\phi = \mathtt{P}_{\geq 0.9}[\mathtt{X\ cl(A)}]$ that states "there exists a block A that can become clear in the next state with a probability ≥ 0.9". The satisfiability relation $s \vDash \phi$ can be rewritten as follows.

$$\begin{aligned}
& s \vDash \mathtt{P}_{\geq 0.9}[\mathtt{X\ cl(A)}] \\
\Leftrightarrow\ & \exists\theta \cdot s \vDash^{\theta} \mathtt{P}_{\geq 0.9}[\mathtt{X\ cl(A)}] \\
\Leftrightarrow\ & \exists\theta \cdot s \vDash^{\theta} P_M^{max}(\{\rho \in Path_{M,s} | \rho(2) \vDash \mathtt{cl(A)}\}) \geq 0.9
\end{aligned}$$

For the ground state $s = \{\mathtt{cl(a)}, \mathtt{on(a,b)}\}$, the above evaluates to *true* with $\theta = \{\mathtt{A/b}\}$ as block b can be clear after taking the action $\mathtt{move(a,fl,b)}$ where fl stands for floor.

It is assumed that the scope of OI-subsumption is within the conjunction. That is, no term inequalities are assumed across different conjunctions. For example, the formula $\mathtt{P}_{\geq 0.7}[\mathtt{X\ cl(A)}] \wedge \mathtt{P}_{\geq 0.95}[\mathtt{F\ on(B,C)}]$ has two conjunctions, and term inequalities such as $\mathtt{A} = \mathtt{B}$ or $\mathtt{A} \neq \mathtt{C}$ do not exist, but $\mathtt{B} \neq \mathtt{C}$ holds under OI-subsumption.

3.3 The relational model checking problem

This section defines the model checking problem for RMDPs, using the definition of RMDP (Sect. 2.2) and relational pCTL (Sect. 3.2). Later, Sects. 4 and 5 will illustrate techniques for solving this model checking problem.

Relational model checking resembles the standard model checking problem (Baier and Katoen 2008). Given a model M and a pCTL formula ϕ, relational model checking computes *all states in M that satisfy* ϕ, denoted by $Sat_M(\phi)$. The significance of relational model checking is that it computes $Sat_M(\phi)$ at a lifted level by using relational states to represent groups of underlying ground states. Hence, relational model checking finds a set of *abstract states* that represents $Sat_M(\phi)$.

Definition 3 Given an RMDP $K = \langle \Sigma, \Delta \rangle$ that defines the underlying MDP $M = \langle S, A, T \rangle$ and a relational pCTL formula ϕ, the relational model checking problem is to determine all ground states $Sat_M(\phi) \subseteq S$ that satisfy ϕ, i.e. $Sat_M(\phi) = \{s \in S | s \vDash \phi\}$. It does so by finding a set of abstract states $Sat_K(\phi)$ in K that represents $Sat_M(\phi)$. Formally,

$$s \in Sat_M(\phi) \Leftrightarrow \exists s' \in Sat_K(\phi).s \preceq_\theta s'$$

In this work, we will solve the relational model checking problem of two types of RMDPs. The first type is the RMDPs that have a finite domain (i.e. finite RMDPs). The second type is a special class of RMDPs that have an infinite domain (i.e. infinite RMDPs).

We discuss the decidability of these two types of RMDPs. The model checking problem for a finite RMDP is decidable. This is because when the domain is finite, the state space is also finite, i.e. the underlying ground MDP is finite. One can thus enumerate all states and collect the states that satisfy the given property. In contrast, an infinite RMDP contains infinitely many states, which makes enumerating all states infeasible. In this case, we focus on a class of infinite RMDPs that have a finite abstraction. More details are given in Sect. 6.

4 PCTL relational Bellman operator

This section defines the *pCTL relational Bellman operator* (pCTL-REBEL), the essential building block for solving the relational model checking problem. Given a pCTL formula $\mathtt{P}_{\bowtie p}[\psi]$ and an RMDP, pCTL-REBEL evaluates a function $V^p : S \to [0, 1]$ that assigns a probability to each RMDP state. A probability $V^p(s)$ represents the probability that state s satisfies the path formula ψ. If the probability is within the bound, i.e. $V^p(s) \bowtie \mathtt{p}$, then state s satisfies $\mathtt{P}_{\bowtie p}[\psi]$ and belongs to the solution set, i.e. $s \in Sat_K(\mathtt{P}_{\bowtie p}[\psi])$.

At this point, it is important to remark that PCTL-REBEL is a variant of REBEL, but does not consider a reward structure. Indeed, REBEL (Kersting et al. 2004) is a model-based relational reinforcement learning technique that operates on a *reward structure* and computes an optimal policy for reaching a set of goal states, which can be seen as a *reward-based* reachability property. However, since we do not consider a reward structure and are interested in *probabilistic* properties, an alternative interpretation of REBEL is required. Section 4.1 introduces an alternative interpretation of the relational Bellman operator. Based on which, Sects. 4.2–4.4 respectively describe in detail the three components of pCTL-REBEL. Section 4.5 then gives an illustration of pCTL-REBEL with an example.

4.1 PCTL relational Bellman operator

Given an RMDP and a pCTL formula $\mathtt{P}_{\bowtie \mathtt{p}}[\psi]$, the task of pCTL-REBEL is to compute a *state probability function* $V^p : S \to [0, 1]$ that assigns a probability to each state. Similar to the original REBEL, pCTL-REBEL takes an initial state probability function V_0^p and iteratively computes V_1^p, V_2^p, etc for a number of steps, depending on the given formula. When the formula has a *step bound* $\mathtt{k}$, then Then pCTL-REBEL is applied for $\mathtt{k}$ times.[1] When the

[1] A step bound is commonly called a finite horizon in AI.

formula is unbounded, pCTL-REBEL is applied for arbitrarily many times until the probabilities converge.

The state probability function V^p is similar to REBEL's state value function V but interprets state values as probabilities rather than as expected rewards. In a similar way, the state-action probability function Q^p is related to REBEL's Q function but interprets state-action values as probabilities. More details will be given later. In order to maintain the connection to the original REBEL and to leave room for extending the present model checking approach to incorporate rewards, we use the same notation V and Q for these functions as in REBEL. For clarification, we add a superscript p to denote that V^p and Q^p interpret values as probabilities. We now formally define these functions and the pCTL relational Bellman operator, following the notations of Kersting et al. (2004).

Definition 4 [cf. Kersting et al. (2004)] A state probability function $V^p : S \rightarrow [0, 1]$ is an ordered set of V^p-rules of the form of $c \leftarrow B$ where B is an abstract state and $c \in [0, 1]$, representing the probability of reaching a goal state from B. The value $V^p(s)$ of a ground state s is assigned by the first rule that subsumes s, i.e. $s \preceq_\theta B$.

Given an abstract goal state G, the initial state probability function V_0^p is defined as

$$1.0 \leftarrow G$$
$$0 \leftarrow \emptyset$$

The first rule expresses that any ground state subsumed by the goal state G, by definition, satisfies G with probability 1. The second rule expresses that any other states that are not captured by the first rule satisfy G with probability 0. The rule of $0 \leftarrow \emptyset$ ensures that all states are assigned a value. Hence, it is often the last V^p-rule to capture the states that are not captured by any previous rules.

Definition 5 [cf. Kersting et al. (2004)] A state-action probability function $Q^p : S \times A \rightarrow [0, 1]$ is an ordered set of Q^p-rules of the form of $c : A \leftarrow B$ where A is an abstract action and B is an abstract state, representing the probability of reaching a goal state when A is taken in B. The value $Q^p(s, a)$ of a ground state s and an action a is assigned by the first rule that subsumes s and a, i.e. $s \preceq_\theta B$ and $a \preceq_\theta A$.

The pCTL-REBEL operator is listed in Eq. (3). By iteratively applying pCTL-REBEL, we compute the state probability functions V_1^p, V_2^p, etc. Notice that pCTL-REBEL is a special case of the original REBEL that sets the discount factor to 1, has no reward structure, and connects a single reward 1 to the target condition.[2] As a result, all V^p values are interpreted as probabilities in [0, 1]. This alternative interpretation allows to capture the probability that a formula is satisfied, which is essential for adapting the original REBEL framework into a model checking setting.

[2] Our work addresses a special case of relational model-based reinforcement learning. More details will be given in Sect. 9.

$$V^p_{t+1}\underbrace{(s)}_{①} = \overbrace{\max_{a\in A(s)} \underbrace{\sum_{s'} T(s,a,s')V^p_t(s')}_{②}}^{③} = \max_{a\in A(s)} Q^p{}_{t+1}(s,a) \tag{3}$$

PCTL-REBEL [Eq. (3)] is implemented by `OneIteration` (Algorithm 1). This algorithm makes use of the following three components.

① `Regression` (Algorithm 2): Deriving the abstract states s in V^p_{t+1}
② Q^p`Rules` (Algorithm 3): Computing $Q^p_{t+1}(s,a)$ for all actions $a \in A(s)$
③ V^p`Rules` (Algorithm 4): Updating V^p_{t+1} by maximizing over $Q^p_{t+1}(s,a)$.

These three components modify the algorithms of Kersting et al. (2004) to provide support to pCTL operators. The modified parts in the algorithms will be marked blue. We now describe these components in detail in Sects. 4.2–4.4, respectively.

Algorithm 1: `OneIteration`. This algorithm implements Equation 3.

Require: V^p_t : state probability function
ψ : pCTL formula [$\mathtt{S}_1$ U $\mathtt{S}_2$] or [X $\mathtt{S}_2$]
Return : V^p_{t+1} : the next probability function

1 $Q^p{}_{t+1} := Q^p$`Rules`$(V^p_t, \mathtt{S}_1)$
2 $V^p_{t+1} := V^p$`Rules`$(Q^p{}_{t+1}, \psi)$

4.2 Logical regression

Logical regression is a standard technique that reasons about abstract transitions at the relational level (Boutilier et al. 2001; Kersting et al. 2004; Sanner and Boutilier 2009). This technique is essential to scalability as it mitigates state explosions by operating on the relational state space instead of the underlying ground state space. Concretely, logical regression searches *backwards* the possible *pre-states* that can reach a given state after taking a number of transitions.

This section extends the standard logical regression to implement Eq. (3) ①. The task is to identify the pre-states that can reach a given state by taking *one transition*, which is related to computing pCTL formulae with a step bound 1. These formulae are of the form of [X $\mathtt{S}_2$], [F$^{\leq 1}\mathtt{S}_2$] or [$\mathtt{S}_1$ U$^{\leq 1}\mathtt{S}_2$]. These three formulae are different in terms of constraints. First, [F$^{\leq 1}\mathtt{S}_2$] is a reachability property that does not impose any constraints. This formula is the basis for the other two formulae. Second, [X $\mathtt{S}_2$] imposes the constraint that "$\mathtt{S}_2$ must be reached after taking *exactly one transition*". Finally, [$\mathtt{S}_1$ U$^{\leq 1}\mathtt{S}_2$] is a *constrained reachability property* that imposes the constraint that "$\mathtt{S}_2$ must be reached by going through *only the states where* $\mathtt{S}_1$ *holds*".

Example 6 To illustrate logical regression, consider an abstract state {on(a, b)} and the following $\delta_{\mathtt{move}_1}$ transition rule (also shown in Fig. 1).

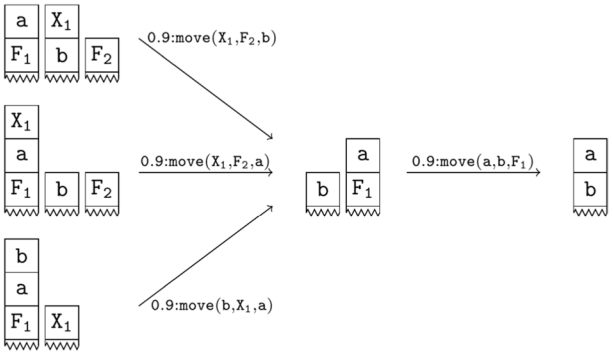

Fig. 3 The figure shows the derived paths by applying two logical regressions on {on(a, b)} with the δ_{move_1} transition rule. The abstract states from left to right reach {on(a, b)} after 2, 1, and 0 steps respectively

$$\delta_{\texttt{move}_1} : \texttt{cl(A)},\texttt{cl(C)},\texttt{on(A, B)} \xleftarrow{\texttt{0.9:move(A,B,C)}} \texttt{cl(A)},\texttt{cl(B)},\texttt{on(A, C)}$$

By applying the substitution $\theta = \{\texttt{A/a}, \texttt{B/b}\}$ to $\delta_{\texttt{move}_1}$, we obtain the following rule. This rules describes that {on(a, b)} can be reached from any ground state that is subsumed by {cl(a), cl(b), on(a, C)} after taking the action move(a, b, C).

$$\texttt{on(a, b)} \xleftarrow{\texttt{0.9:move(a,b,C)}} \texttt{cl(a)}, \texttt{cl(b)}, \texttt{on(a, C)}$$

Logical regression can be applied multiple times. For example, by applying twice the $\delta_{\texttt{move}_1}$ transition rule, we obtain all abstract states that can reach {on(a, b)} within 2 steps, as shown in Fig. 3.

Logical regression is implemented in `Regression` (Algorithm 2) that identifies all states that can reach a goal state after taking one transition. `Regression` generalizes the `WEAKESTPRE` algorithm in REBEL (Kersting et al. 2004) in two ways (marked blue, line 6–9). First, it provides support to *until* formulae $[\texttt{S}_1\ \texttt{U}\ \texttt{S}_2]$ by ensuring that all pre-states satisfy $\texttt{S}_1$. Second, it provides support to infinite RMDPs by filtering out the pre-states that exceed a given state bound b. More details will be given in Sect. 6.3.[3]

[3] The state bound $b \in \mathbb{N}$ is for infinite RMDPs. More details in Sect. 6.

Algorithm 2: `Regression`. This algorithm implements Equation 3 ①. All returned states must be subsumed by $\mathtt{S}_1$, and can reach $\mathtt{S}_2$ by following the δ transition rule. All returned states must be legal and respect OI-subsumption.

Require: δ : transition rule $\delta = H_i \xleftarrow{p_i:\alpha} B$
$\mathtt{S}_1$: all returned states must be subsumed by $\mathtt{S}_1$
$\mathtt{S}_2$: goal state
Return : `PreS` : states that can reach $\mathtt{S}_2$ by following δ

1 `PreS` := $\emptyset$
2 **foreach** $\mathtt{S}'_2 \subseteq \mathtt{S}_2$ *and* $\mathtt{H}'_i \subseteq H_i$ *s.t.* $\theta = unifier(\mathtt{S}'_2, \mathtt{H}'_i)$ *exists* **do**
3 $\mathtt{S} := (\mathtt{S}_2\theta \backslash \mathtt{H}'_i\theta) \cup B\theta$ `// identify a pre-state S`
4 **foreach** $l \in (\mathtt{S}_2\theta \backslash \mathtt{H}'_i\theta)$ *and* $l' \in H_i\theta \cup B\theta$ *s.t.* $unifier(l, l')$ *exists* **do**
5 add $l \neq l'$ to S `// add object inequalities to S`
6 $\mathtt{S}_{\mathtt{pre}} := mgs(\mathtt{S}, \mathtt{S}_1)$ `// apply the S1 constraint`
7 **foreach** $\mathtt{s}_{\mathtt{pre}} \in \mathtt{S}_{\mathtt{pre}}$ **do**
8 **if** $|terms(\mathtt{s}_{\mathtt{pre}})| \leq b$ **then** add $\mathtt{s}_{\mathtt{pre}}$ to `PreS` `// apply the state bound`[3]
9

4.3 Q^p-Rules generation

Given a state probability function V^p, this section computes Q^p probabilities for state-action pairs (cf. Eq. (3) ②). $Q^p(s, a)$ denotes the probability that a formula is satisfied by a path that starts from s and has the first action a. Similar to the state probability function V^p, the Q^p function is an ordered set of Q^p-rules. These Q^p-rules are an intermediate representation, from which we can compute the next state probability function (cf. Sect. 4.4).

Deriving a Q^p function is complex for two reasons. First, we must consider all transition rules that indicate different action postconditions. For example, action `move/3` has two possible outcomes, namely succeeding and failing. As could be expected, it is not sufficient to consider only the case where the action always succeeds as in Fig. 3. We must consider all transition rules in order to correctly calculate the Q^p-rules. Second, an abstract action can *diverge* and produce multiple preconditions. For example, Fig. 3 shows three different preconditions are derived by applying different substitutions to the $\delta_{\mathtt{move}_i}$ transition. It is required to consider all these preconditions. Fortunately, to resolve these issues, we could follow the procedure provided by the original REBEL, with limited adaptions.

Q^p`Rules` (Algorithm 3) implements the procedure of computing a set of Q^p-rules (cf. Eq. (3) ②). The process is as the follows. First, for each transition rule and current states in V^p_t, Q^p`Rules` computes a set of *partial rules* (cf. line 2–5). This procedure results in multiple sets of partial rules, and each set considers one single postcondition of the action. Then, we must combine these partial rule sets to get a complete Q^p-rule set. This is done by unifying the partial rules (cf. line 6–14). Finally, Q^p`Rules` returns a set of complete Q^p-rules.

Q^p`Rules` modifies the `QRULES` algorithm in REBEL (Kersting et al. 2004) in two ways (marked blue, line 4, 5 and 12). First, it provides support to the *until* formulae [$\mathtt{S}_1$ U $\mathtt{S}_2$] by passing an extra parameter $\mathtt{S}_1$ to Algorithm 2 (cf. Sect. 4.2). Second, it achieves the probabilistic interpretation of the Bellman operator by discarding the reward component and setting the discount factor to 1.

Algorithm 3: Q^p`Rules`. This algorithm implements Equation 3 ②. All returned Q^p-rules must be legal and respect OI-subsumption. All states in the returned Q^p-rules must be subsumed by $\mathtt{S}_1$.

Require: V_t^p : state probability function
$\mathtt{S}_1$: all states in the returned Q^p-rules must be subsumed by $\mathtt{S}_1$
Return : ${Q^p}_{t+1}$: the next state-action probability function

1 ${Q^p}_{t+1} := \emptyset$
/* fix an action α */
2 **foreach** $H_i \xleftarrow{p_i:\alpha} B$ *for* α **do**
3 **foreach** $v^p \leftarrow V \in V_t^p$ **do**
4 $\mathtt{PreS} := \mathtt{Regression}(H_i \xleftarrow{p_i:\alpha} B, \mathtt{S}_1, V)$
/* derive a subset of partial rules */
5 $partial\,Q^p := \{q^p : \alpha\theta \leftarrow S | S \in \mathtt{PreS} \text{ and } q^p = p_i \times v^p\}$
6 **if** ${Q^p}_{t+1} = \emptyset$ **then** ${Q^p}_{t+1} := partial\,Q^p$
7 **else**
8 $new\,Q^p := \emptyset$
9 **for** *all pairs* $q_1^p : A_1 \leftarrow S_1 \in {Q^p}_{t+1}$ *and*
10 $q_2^p : A_2 \leftarrow S_2 \in partial\,Q^p$ **do**
/* unify the partial rules */
11 **foreach** $A \leftarrow S = mgs(A_1 \leftarrow S_1, A_2 \leftarrow S_2)$ **do**
12 $q^p := q_1^p + q_2^p$
13 add $q^p : A \leftarrow S$ to $new\,Q^p$
14 ${Q^p}_{t+1} := new\,Q^p$

4.4 V^p-Rules generation

This section calculates the new state probability function V_{t+1}^p, given ${Q^p}_{t+1}$, by maximizing over the actions (cf. Eq. (3) ③). Recall that ${Q^p}_{t+1}$ is an ordered set of Q^p-rules of the form of $q^p : A \leftarrow S$. The task is to derive V_{t+1}^p, an ordered set of V^p-rules of the form of $v^p \leftarrow S$.

Trivially, turning ${Q^p}_{t+1}$ into V_{t+1}^p takes three steps. First, the Q^p-rules must be sorted such that a rule connected to a high probability has a high priority as we are interested in the *maximum* probability [as defined by Eq. (2)]. Second, the redundant Q^p-rules must be removed. A Q^p-rule is redundant if it is subsumed by another Q^p-rule that has a higher priority. Third, the remaining Q^p-rules are turned into V^p-rules by removing the action in the rule.

V^p`Rules` (Algorithm 4) implements Eq. (3) ③. The process is as follows. First, the Q^p-rules are ordered decreasingly so that a state is always assigned a *maximum* probability (line 2). Second, to remove redundant Q^p-rules, an *absorbing rule* is required when the formula has an absorbing goal in that no more transitions occur after the goal is reached. Hence, given an absorbing goal, any rule concerning transitions that start from the goal is redundant and should be removed. Concretely, an *until* formula $[\mathtt{S}_1\ \mathtt{U}\ \mathtt{S}_2]$ has an absorbing goal $\mathtt{S}_2$ such that an execution stops once $\mathtt{S}_2$ is reached. Hence, an absorbing rule $1.0 \leftarrow \mathtt{S}_2$ must be inserted to the beginning of the Q^p-rules. On the other hand, a *next* formula $[\mathtt{X}\ \mathtt{S}_2]$ does not have an absorbing goal as it is possible to follow exactly one transition from $\mathtt{S}_2$ to another state where $\mathtt{S}_2$ may or may not hold. Therefore, no absorbing rules are inserted. Finally, redundant rules in the ordered set are removed (line 6–10).

V^pRules generalizes the VRULES algorithm in REBEL (Kersting et al. 2004) to handle non-absorbing *next* formulae [X S_2] as the original REBEL considers only absorbing goals. The generalized part is marked blue (i.e. line 3–5).

Algorithm 4: V^pRules. This function implements Equation 3 ③. All returned V^p-rules must be legal and respect OI-subsumption.

Require: $Q^p{}_{t+1}$: state-action probability function
ψ : pCTL formula [S_1 U S_2] or [X S_2]
Return : V^p_{t+1} : the next state probability function

1 $V^p_{t+1} := \emptyset$
2 sort $Q^p{}_{t+1}$ in decreasing order of Q^p probability
/* Add absorbing Q^p-rules */
3 **if** ψ = S_1 $U^{\leq k}$ S_2 **then** $Q^p{}_{abs} := \{1.0 : \emptyset \leftarrow S | S \in S_2\}$
4 **else** $Q^p{}_{abs} := \emptyset$
5 $Q^p{}_{t+1} := Q^p{}_{abs} + Q^p{}_{t+1}$ // attach absorbing Q^p-rules to the top of $Q^p{}_{t+1}$
6 **while** $Q^p{}_{t+1} \neq \emptyset$ **do**
7 remove the top Q^p-rule $d : A \leftarrow B$ from their $Q^p{}_{t+1}$
8 **if** *no other rule* $d : A' \leftarrow B'$ *in* $Q^p{}_{t+1}$ *exists s.t.* B' *subsumes* B **then**
9 add $d \leftarrow B$ to V^p_{t+1}
/* remove redundant Q^p-rules */
10 remove all rules $d'' \leftarrow B''$ from $Q^p{}_{t+1}$ s.t. B subsumes B''

4.5 PCTL-REBEL illustration

This section illustrates pCTL-REBEL (cf. Eq. (3)), namely, taking a state probability function V^p_t to compute the next state probability function V^p_{t+1}. Clearly, different pCTL formulae require different numbers of iterations. That is, [X S_2] requires one iteration, [S_1 $U^{\leq k}$ S_2] requires k iterations, and [S_1 U S_2] requires an arbitrary number iterations to obtain an accurate enough approximation. For simplicity, we illustrate with path formulae (without probabilities) that require one iteration. Section 5 will cover the full pCTL language.

Formula 1 ψ_1 = [X on(a, b)]: find all states that reach {on(a, b)} after 1 step.
Formula 2 ψ_2 = [on(c, d) $U^{\leq 1}$ on(a, b)]: find all states that reach {on(a, b)} within 1 step by going through only the states where {on(c, d)} holds.

We consider the $\delta_{\texttt{move}}$ transition in the blocks world (also in Fig. 1).

$$\delta_{\texttt{move}_1} : \texttt{cl(A),cl(C),on(A,B)} \xleftarrow{0.9:\texttt{move(A,B,C)}} \texttt{cl(A),cl(B),on(A,C)}$$

$$\delta_{\texttt{move}_2} : \texttt{cl(A),cl(B),on(A,C)} \xleftarrow{0.1:\texttt{move(A,B,C)}} \texttt{cl(A),cl(B),on(A,C)}$$

4.5.1 PCTL-REBEL on Formula 1: [X on(a, b)]

For the formula ψ_1 = [X on(a, b)], the initial function V^p_0 is

Table 1 Different combinations of transition rules and states result in different partial Q^p-rules $\langle 1a\rangle$–$\langle 2e\rangle$. The states on the leftmost column comes from V_0^p

	$\delta_{\texttt{move}_1}$	$\delta_{\texttt{move}_2}$
on(a, b)	$\langle 1a\rangle\langle 1b\rangle\langle 1c\rangle\langle 1d\rangle$	$\langle 2a\rangle\langle 2b\rangle\langle 2c\rangle\langle 2d\rangle$
$\emptyset$	$\langle 1e\rangle$	$\langle 2e\rangle$

$1.0 \leftarrow \texttt{on(a, b)}.$
$0.0 \leftarrow \emptyset.$

①`Regression` Given the initial state probability function V_0^p, to obtain all possible pre-states, `Regression` is called with all combinations of transition rules and V^p-rules, e.g. $\texttt{Regression}(\delta_{\texttt{move}_1}, \emptyset, \texttt{on(a, b)})$.[4] This results in two sets of partial Q^p-rules such that each set considers one outcome of the `move` action. The resulting partial Q^p-rules are listed below. The $\langle 1\cdot\rangle$ rules correspond to the successful outcome (i.e. $\delta_{\texttt{move}_1}$) and the $\langle 2\cdot\rangle$ rules correspond to the unsuccessful outcome (i.e. $\delta_{\texttt{move}_2}$). Table 1 shows the the corresponding transitions and states. All partial Q^p-rules respect OI-subsumption.

$$\langle 1a\rangle\ 0.9 : \texttt{move(a, b, Z)} \leftarrow \texttt{cl(a)}, \texttt{cl(b)}, \texttt{on(a, Z)}$$
$$\langle 1b\rangle\ 0.9 : \texttt{move(X, a, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(a)}, \texttt{on(X, Z)}, \texttt{on(a, b)}$$
$$\langle 1c\rangle\ 0.9 : \texttt{move(X, Y, a)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, a)}, \texttt{on(a, b)}$$
$$\langle 1d\rangle\ 0.9 : \texttt{move(X, Y, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)}, \texttt{on(a, b)}$$
$$\langle 1e\rangle\ 0.0 : \texttt{move(X, Y, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)}$$
$$\langle 2a\rangle\ 0.1 : \texttt{move(a, Y, b)} \leftarrow \texttt{cl(a)}, \texttt{cl(Y)}, \texttt{on(a, b)}$$
$$\langle 2b\rangle\ 0.1 : \texttt{move(X, a, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(a)}, \texttt{on(X, Z)}, \texttt{on(a, b)}$$
$$\langle 2c\rangle\ 0.1 : \texttt{move(X, Y, a)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, a)}, \texttt{on(a, b)}$$
$$\langle 2d\rangle\ 0.1 : \texttt{move(X, Y, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)}, \texttt{on(a, b)}$$
$$\langle 2e\rangle\ 0.0 : \texttt{move(X, Y, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)}$$

② Q^p`Rules` To compute the Q^p-rules, the two sets of partial Q^p-rules must be combined. To do so, we consider all possible combinations of $\langle 1\cdot\rangle$ and $\langle 2\cdot\rangle$. The resulting Q^p-rules are listed as follows, along with how they are created. All rules respect OI-subsumption.

[4] The second argument is $\emptyset$ as the `X` operator is not absorbing.

$$
\begin{aligned}
\langle 1\rangle\ 1.0 &: \texttt{move(X, a, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(a)}, \texttt{on(X, Z)}, \texttt{on(a, b)} && \langle 1b\rangle{+}\langle 2b\rangle\\
\langle 2\rangle\ 1.0 &: \texttt{move(X, Y, a)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, a)}, \texttt{on(a, b)} && \langle 1c\rangle{+}\langle 2c\rangle\\
\langle 3\rangle\ 1.0 &: \texttt{move(X, Y, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)}, \texttt{on(a, b)} && \langle 1d\rangle{+}\langle 2d\rangle\\
\langle 4\rangle\ 0.9 &: \texttt{move(a, b, Z)} \leftarrow \texttt{cl(a)}, \texttt{cl(b)}, \texttt{on(a, Z)} && \langle 1a\rangle{+}\langle 2e\rangle\\
\langle 5\rangle\ 0.9 &: \texttt{move(X, a, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(a)}, \texttt{on(X, Z)}, \texttt{on(a, b)} && \langle 1b\rangle{+}\langle 2e\rangle\\
\langle 6\rangle\ 0.9 &: \texttt{move(X, Y, a)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, a)}, \texttt{on(a, b)} && \langle 1c\rangle{+}\langle 2e\rangle\\
\langle 7\rangle\ 0.9 &: \texttt{move(X, Y, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)}, \texttt{on(a, b)} && \langle 1d\rangle{+}\langle 2e\rangle\\
\langle 8\rangle\ 0.1 &: \texttt{move(a, Y, b)} \leftarrow \texttt{cl(a)}, \texttt{cl(Y)}, \texttt{on(a, b)} && \langle 1e\rangle{+}\langle 2a\rangle\\
\langle 9\rangle\ 0.1 &: \texttt{move(X, a, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(a)}, \texttt{on(X, Z)}, \texttt{on(a, b)} && \langle 1e\rangle{+}\langle 2b\rangle\\
\langle 10\rangle\ 0.1 &: \texttt{move(X, Y, a)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, a)}, \texttt{on(a, b)} && \langle 1e\rangle{+}\langle 2c\rangle\\
\langle 11\rangle\ 0.1 &: \texttt{move(X, Y, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)}, \texttt{on(a, b)} && \langle 1e\rangle{+}\langle 2d\rangle\\
\langle 12\rangle\ 0.0 &: \texttt{move(X, Y, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)} && \langle 1e\rangle{+}\langle 2e\rangle
\end{aligned}
$$

The Q^p-rules must be ordered by their probabilities as above. The rules $\langle 1\rangle$–$\langle 3\rangle$ are interchangeable as they have the same probability. Similarly, $\langle 4\rangle$–$\langle 7\rangle$ and $\langle 8\rangle$–$\langle 11\rangle$ are interchangeable, respectively.

③ V^p`Rules` Now we can derive the new V^p-rules by removing redundant Q^p-rules and dropping the action components. The resulting V^p-rules are shown below where the numbering inherits the one of the Q^p-rules. Rules $\langle 5\rangle$-$\langle 7\rangle$ are redundant because they are subsumed by $\langle 1\rangle$-$\langle 3\rangle$, respectively. Similarly, rules $\langle 9\rangle$-$\langle 11\rangle$ are redundant as they are subsumed by $\langle 1\rangle$–$\langle 3\rangle$, respectively.

$$
\begin{aligned}
\langle 1\rangle\ 1.0 &\leftarrow \texttt{cl(X)}, \texttt{cl(a)}, \texttt{on(X, Z)}, \texttt{on(a, b)}\\
\langle 2\rangle\ 1.0 &\leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, a)}, \texttt{on(a, b)}\\
\langle 3\rangle\ 1.0 &\leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)}, \texttt{on(a, b)}\\
\langle 4\rangle\ 0.9 &\leftarrow \texttt{cl(a)}, \texttt{cl(b)}, \texttt{on(a, Z)}\\
\langle 8\rangle\ 0.1 &\leftarrow \texttt{cl(a)}, \texttt{cl(Y)}, \texttt{on(a, b)}\\
\langle 12\rangle\ 0.0 &\leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)}
\end{aligned}
$$

Given $\psi_1 = [\texttt{X on(a, b)}]$, and the initial state probability function V_0^p, we have applied pCTL-REBEL and obtained V_1^p, which assigns to all states a maximum probability of reaching `on(a, b)` after exactly one step.

4.5.2 PCTL-REBEL on Formula 2: [on(c, d) $\mathtt{U}^{\leq 1}$ on(a, b)]

For the formula $\psi_2 = [\texttt{on(c, d)}\ \mathtt{U}^{\leq 1}\texttt{on(a, b)}]$, the initial probability function V_0^p is

$$
\begin{aligned}
1.0 &\leftarrow \texttt{on(a, b)}\\
0.0 &\leftarrow \emptyset
\end{aligned}
$$

① `Regression` Given the initial state probability function V_0^p, to obtain all possible pre-states, `Regression` is called with all combinations of transition rules and V^p-rules, e.g. `Regression`($\delta_{\texttt{move}_1}$, `on(c, d)`, `on(a, b)`). This results in two sets of partial Q^p-rules such that each set considers one outcome of the `move` action. We list some of the resulting partial Q^p-rules below. The $\langle 1\cdot\rangle$ rules correspond to the successful outcome (i.e. $\delta_{\texttt{move}_1}$) and the

Table 2 Different combinations of transition rules and states result in different partial Q^p-rules $\langle 1a\rangle$-$\langle 2o\rangle$

	$\delta_{\texttt{move}_1}$	$\delta_{\texttt{move}_2}$
on(a, b)	$\langle 1a\rangle$-$\langle 1k\rangle$	$\langle 2a\rangle$-$\langle 2k\rangle$
∅	$\langle 1l\rangle$-$\langle 1o\rangle$	$\langle 2l\rangle$-$\langle 2o\rangle$

The states on the leftmost column comes from V_0^p

$\langle 2\cdot\rangle$ rules correspond to the unsuccessful outcome (i.e. $\delta_{\texttt{move}_2}$). Table 2 shows the the corresponding transitions and states. All partial Q^p-rules respect OI-subsumption.

$$
\begin{aligned}
&\langle 1a\rangle\ 0.9 : \texttt{move(a, b, c)} \leftarrow \texttt{cl(a)}, \texttt{cl(b)}, \texttt{on(a, c)}, \texttt{on(c, d)}\\
&\langle 1b\rangle\ 0.9 : \texttt{move(a, b, Z)} \leftarrow \texttt{cl(a)}, \texttt{cl(b)}, \texttt{on(a, Z)}, \texttt{on(c, d)}\\
&\langle 1c\rangle\ 0.9 : \texttt{move(c, a, d)} \leftarrow \texttt{cl(a)}, \texttt{cl(c)}, \texttt{on(a, b)}, \texttt{on(c, d)}\\
&\langle 1d\rangle\ 0.9 : \texttt{move(c, Y, d)} \leftarrow \texttt{cl(c)}, \texttt{cl(Y)}, \texttt{on(a, b)}, \texttt{on(c, d)}\\
&\quad\ldots\\
&\langle 1l\rangle\ 0.0 : \texttt{move(X, c, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(c)}, \texttt{on(X, Z)}, \texttt{on(c, d)}\\
&\langle 1m\rangle\ 0.0 : \texttt{move(c, Y, d)} \leftarrow \texttt{cl(c)}, \texttt{cl(Y)}, \texttt{on(c, d)}\\
&\langle 1n\rangle\ 0.0 : \texttt{move(X, Y, c)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, c)}, \texttt{on(c, d)}\\
&\langle 1o\rangle\ 0.0 : \texttt{move(X, Y, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)}, \texttt{on(c, d)}\\
&\langle 2a\rangle\ 0.1 : \texttt{move(c, a, d)} \leftarrow \texttt{cl(a)}, \texttt{cl(c)}, \texttt{on(a, b)}, \texttt{on(c, d)}\\
&\langle 2b\rangle\ 0.1 : \texttt{move(c, Y, d)} \leftarrow \texttt{cl(c)}, \texttt{cl(Y)}, \texttt{on(a, b)}, \texttt{on(c, d)}\\
&\quad\ldots\\
&\langle 2m\rangle\ 0.0 : \texttt{move(c, Y, d)} \leftarrow \texttt{cl(c)}, \texttt{cl(Y)}, \texttt{on(c, d)}\\
&\langle 2n\rangle\ 0.0 : \texttt{move(X, Y, c)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, c)}, \texttt{on(c, d)}\\
&\langle 2o\rangle\ 0.0 : \texttt{move(X, Y, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)}, \texttt{on(c, d)}
\end{aligned}
$$

② Q^p`Rules` To compute the Q^p-rules, the two sets of partial Q^p-rules must be combined. To do so, we consider all possible combinations of $\langle 1\cdot\rangle$ and $\langle 2\cdot\rangle$. Some of the resulting Q^p-rules are listed below, along with how they are created. All rules respect OI-subsumption.

$$
\begin{aligned}
&\langle 1\rangle\ 1.0 : \texttt{move(c, a, d)} \leftarrow \texttt{cl(a)}, \texttt{cl(c)}, \texttt{on(a, b)}, \texttt{on(c, d)} && \langle 1c\rangle{+}\langle 2a\rangle\\
&\langle 2\rangle\ 1.0 : \texttt{move(c, Y, d)} \leftarrow \texttt{cl(c)}, \texttt{cl(Y)}, \texttt{on(a, b)}, \texttt{on(c, d)} && \langle 1d\rangle{+}\langle 2b\rangle\\
&\quad\ldots\\
&\langle 10\rangle\ 0.9 : \texttt{move(a, b, c)} \leftarrow \texttt{cl(a)}, \texttt{cl(b)}, \texttt{on(a, c)}, \texttt{on(c, d)} && \langle 1a\rangle{+}\langle 2n\rangle\\
&\langle 11\rangle\ 0.9 : \texttt{move(a, b, Z)} \leftarrow \texttt{cl(a)}, \texttt{cl(b)}, \texttt{on(a, Z)}, \texttt{on(c, d)} && \langle 1b\rangle{+}\langle 2o\rangle\\
&\quad\ldots\\
&\langle 33\rangle\ 0.0 : \texttt{move(c, Y, d)} \leftarrow \texttt{cl(c)}, \texttt{cl(Y)}, \texttt{on(c, d)} && \langle 1m\rangle{+}\langle 2m\rangle\\
&\langle 34\rangle\ 0.0 : \texttt{move(X, Y, c)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, c)}, \texttt{on(c, d)} && \langle 1n\rangle{+}\langle 2n\rangle\\
&\langle 35\rangle\ 0.0 : \texttt{move(X, Y, Z)} \leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)}, \texttt{on(c, d)} && \langle 1o\rangle{+}\langle 2o\rangle
\end{aligned}
$$

The Q^p-rules must be ordered by their probabilities as above.

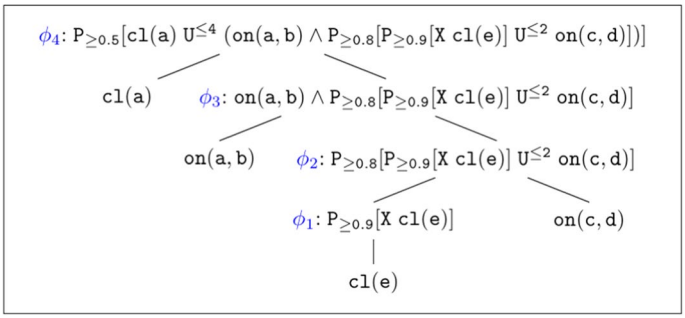

Fig. 4 The parse tree of $\phi_{\texttt{nested}}$. Each inner node of the parse tree is annotated with a subformula ϕ_i and handles one operator. ϕ_1 handles X, ϕ_2 handles $U^{\leq 2}$, ϕ_3 handles $\wedge$ and ϕ_4 handles $U^{\leq 4}$

③ V^p`Rules` Now we can derive the new V^p-rules. Since [on(a, b)] is absorbing, the following absorbing rule must be inserted to the beginning of the Q^p-rule set.

$$\langle 0 \rangle \; 1.0 : \emptyset \leftarrow \texttt{on(a, b)}$$

After removing redundant Q^p-rules and dropping the action components in the rules, we obtain the resulting V^p-rules below. The numbering of the V^p-rules inherits the one of the Q^p-rules. Most redundant rules are subsumed by the absorbing rule $\langle 0 \rangle$.

$$\begin{aligned}
\langle 0 \rangle \; 1.0 &\leftarrow \texttt{on(a, b)} \\
\langle 10 \rangle \; 0.9 &\leftarrow \texttt{cl(a)}, \texttt{cl(b)}, \texttt{on(a, c)}, \texttt{on(c, d)} \\
\langle 11 \rangle \; 0.9 &\leftarrow \texttt{cl(a)}, \texttt{cl(b)}, \texttt{on(a, Z)}, \texttt{on(c, d)} \\
\langle 33 \rangle \; 0.0 &\leftarrow \texttt{cl(c)}, \texttt{cl(Y)}, \texttt{on(c, d)} \\
\langle 34 \rangle \; 0.0 &\leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, c)}, \texttt{on(c, d)} \\
\langle 35 \rangle \; 0.0 &\leftarrow \texttt{cl(X)}, \texttt{cl(Y)}, \texttt{on(X, Z)}, \texttt{on(c, d)}
\end{aligned}$$

5 Main contribution: a relational model checker

This section introduces the main algorithm in this paper, the relational model checking algorithm, that solves the relational model checking problem (Sect. 3.3). Formally, given a pCTL formula ϕ and an RMDP K, the relational model checker identifies the set $Sat_K(\phi)$ of abstract states that represent all ground states that satisfy ϕ. Since the relational model checker is based on PCTL-REBEL (Sect. 4), it operates at the relational level.

Different from Sect. 4 that computes one single iteration, this section allows for full relational pCTL formulae that requires multiple iterations and can be *nested*. It is standard practice to represent a nested pCTL formula as a parse tree (Baier and Katoen 2008). In a parse tree, each leaf node is an abstract state and each inner node contains exactly one operator.

Example 7 A nested relational pCTL formula example is as follows.

$$\phi_{\texttt{nested}} = \texttt{P}_{\geq 0.5}[\texttt{cl(a)}\ \texttt{U}^{\leq 4}(\texttt{on(a, b)} \wedge \texttt{P}_{\geq 0.8}[\texttt{P}_{\geq 0.9}[\texttt{X cl(e)}]\ \texttt{U}^{\leq 2}\texttt{on(c, d)}])]$$

The nested formula $\phi_{\texttt{nested}}$ is complex at first sight, however, it can be represented as a *parse tree* in Fig. 4. A state s satisfies $\phi_{\texttt{nested}}$ if and only if s satisfies all the following three

conditions. (1) A path starting from s must reach $\{\texttt{on(a,b)}\}$ within 4 steps with a probability greater than or equal to 0.5 by going through the states where $\{\texttt{cl(a)}\}$ holds. (2) Then, the path must reach $\{\texttt{on(c,d)}\}$ within 2 steps with a probability greater than or equal to 0.8 by going through the states that satisfy $\texttt{P}_{\geq 0.9}[\texttt{X cl(e)}]$. (3) The states that satisfy $\texttt{P}_{\geq 0.9}[\texttt{X cl(e)}]$ are the ones that can transition to $\{\texttt{cl(e)}\}$ with a probability greater than or equal to 0.9 after exactly one step.

PCTL-REBEL always use a parse tree to evaluate a given formula. A parse tree is recursively evaluated *upwards*. That is, an inner node considers its child leaf nodes to evaluate a subformula ϕ_i, resulting in a set of states $Sat(\phi_i)$. Then, the inner node *folds* the sub-tree so that its parent node can be activated. The final set of states $Sat(\phi)$ is produced by the root. For example, in Fig. 4, an intermediate state set $Sat(\phi_i)$ is used to evaluate $Sat(\phi_{i+1})$. The state set $Sat(\phi_4)$ is returned as the solution.

Section 5.1 defines the pCTL-REBEL model checking algorithm, which is composed of 3 mutually recursive algorithms. Then, Sect. 5.2 gives an overview of the properties of the model checking algorithm.

5.1 The pCTL-REBEL model checking algorithm

The pCTL-REBEL model checking algorithm reforms the satisfiability relation $\models^{\theta}$ (cf. Sect. 3.2). Given an RMDP K and a pCTL formula ϕ, the pCTL-REBEL model checker $Sat_K(\phi)$, i.e. all states in K that satisfy ϕ. The model checker consists of 3 algorithms `Check`, `CheckUntil` and `CheckNext` that call one another recursively. In particular, `Check` is the main algorithm and is mutually recursive with the other two algorithms. `CheckUntil` handles the *until* formulae of the form of $\texttt{P}_{\bowtie p}[\phi_1\ U^{\leq k}\phi_2]$. `CheckNext` handles the *next* formulae of the form of $\texttt{P}_{\bowtie p}[\phi_2]$.

Given a formula ϕ, `Check` (Algorithm 5) evaluates ϕ in a recursive fashion as follows. If ϕ is a relation atom l (resp. $\neg l$), it returns a single abstract state $\{l\}$ (resp. $\{\neg l\}$) (line 1–2). If ϕ is a conjunction of the form of $\phi_1 \wedge \phi_2$, it first computes ϕ_1 and ϕ_2 separately to get two sets of abstract states $Sat1$ and $Sat2$ (line 4–5). Then, for each possible pair of abstract states $s_1 \in Sat1$ and $s_2 \in Sat2$, it collects all maximally general specializations $mgs(s_1, s_2)$ (line 6). Since all $mgs(s_1, s_2)$ are OI-subsumed by s_1 and s_2 by definition, they automatically satisfy $\phi_1 \wedge \phi_2$. If ϕ is a disjunction of the form of $\phi_1 \vee \phi_2$, it returns the union of the solutions to ϕ_1 and ϕ_2 (line 7–8). If ϕ is an *until* formula, it calls `CheckUntil` (Algorithm 6) (line 9–10). If ϕ is a *next* formula, it calls `CheckNext` (Algorithm 7) (line 11–12).

Algorithm 5: `Check`. The main algorithm to solve the relational model checking problem. This algorithm implements the satisfiability relation (Section 3.2). `Check` is mutually recursive with `CheckUntil` and `CheckNext`.

Require: ϕ : pCTL formula
Return : $Sat_K(\phi)$: abstract states that satisfy ϕ

1 **if** $\phi = l$ **then** $Sat_K(\phi) = \{1\}$
2 **else if** $\phi = \neg l$ **then** $Sat_K(\phi) = \{\neg l\}$
3 **else if** $\phi = \phi_1 \wedge \phi_2$ **then**
4 $\quad Sat_1 =$ `Check`(ϕ_1)
5 $\quad Sat_2 =$ `Check`(ϕ_2)
6 $\quad Sat_K(\phi) = \{mgs(s1, s2) \mid s1 \in Sat_1, s2 \in Sat_2\}$
7 **else if** $\phi = \phi_1 \vee \phi_2$ **then**
8 $\quad Sat_K(\phi) =$ `Check`$(\phi_1) \cup$ `Check`(ϕ_2)
9 **else if** $\phi = \mathtt{P}_{\bowtie p}[\phi_1 \mathtt{U}^{\leq k} \phi_2]$ **then**
10 $\quad Sat_K(\phi) =$ `CheckUntil`(ϕ)
11 **else if** $\phi = \mathtt{P}_{\bowtie p}[\mathtt{X}\ \phi_2]$ **then**
12 $\quad Sat_K(\phi) =$ `CheckNext`(ϕ)

`CheckUntil` (Algorithm 6) is mutually recursive with `Check` (Algorithm 5). It computes a formula of the form of $\mathtt{P}_{\bowtie p}[\phi_1\ U^{\leq k} \phi_2]$ or $\mathtt{P}_{\bowtie p}[\phi_1\ U^{\leq k} \phi_2]$ where ϕ_1 and ϕ_2 are pCTL formulae. To begin the process, `CheckUntil` computes ϕ_1 and ϕ_2 separately to get two sets of abstracts states (line 1–2). Then, it sets the step bound for pCTL-REBEL (line 3). When the given formula has a bounded $\mathtt{U}^{\leq k}$ operator, the step bound is `k` . Otherwise, the step bound is set to infinity and an arbitrary number of pCTL-REBEL iterations are applied until convergence (line 5–11). The convergence condition is twofold. First, the abstract states in the probability function do not change, i.e. states in V_t^p and V_{t-1}^p are the same. Second, the state probabilities have converged with respect to a given threshold ϵ, i.e. $\max_{s \in V_t^p} |V_t^p(s) - V_{t-1}^p(s)| < \epsilon$. Then, `Check-Until` collects and returns the abstract states in V_t^p that satisfy the given probability threshold $\bowtie$ p (line 12–14).

Algorithm 6: `CheckUntil`. `CheckUntil` is mutually recursive with `Check`.

Require: ϕ : pCTL formula $\mathtt{P}_{\bowtie \mathtt{p}}[\phi_1 \mathtt{U}^{\leq \mathtt{k}} \phi_2]$ or $\mathtt{P}_{\bowtie \mathtt{p}}[\phi_1 \mathtt{U} \phi_2]$
Return : $Sat_K(\phi)$: abstract states that satisfy ϕ

1 $\mathtt{S}_1 := \mathtt{Check}(\phi_1)$
2 $\mathtt{S}_2 := \mathtt{Check}(\phi_2)$
3 **if** k *is not given* **then** $\mathtt{k} := \infty$ // set k as infinity for unbounded U
4
5 $V_0^p := \{1.0 \leftarrow s | s \in \mathtt{S}_2\} \cup \{0 \leftarrow \emptyset\}$ // Initialize V_0^p
6 $t := 0$ // t is the number of iterations so far
7 $\epsilon' := \infty$ // ϵ' is the distance between V_t^p and V_{t+1}^p
8 **while** $t < \mathtt{k}$ *or* $\epsilon' \geq \epsilon$ **do**
9 $V_{t+1}^p := \mathtt{OneIteration}(V_t^p, \mathtt{S}_1 \mathtt{U}^{\leq \mathtt{k}} \mathtt{S}_2)$
10 $\epsilon' := V_{t+1}^p$ - V_t^p
11 $t := t + 1$
12 $Sat_K(\phi) := \emptyset$
13 **foreach** $p_i \leftarrow s_i \in V_t^p$ **do**
14 **if** $p_i \bowtie \mathtt{p}$ **then** add s_i to $Sat_K(\phi)$

`CheckNext` (Algorithm 7) is mutually recursive with `Check` (Algorithm 5). It computes a formula of the form of $\phi = \mathtt{P}_{\bowtie \mathtt{p}}[\mathtt{X}\ \phi_2]$ where ϕ_2 is another pCTL formula. `CheckNext` is a similar but simpler than `CheckUntil`. It performs one single value iteration to obtain the probability function V_1^p (line 2–3). Then it collects and returns the abstract states in V_1^p that satisfy the given probability threshold $\bowtie \mathtt{p}$ (line 4–6). `CheckNext` needs only one iteration as it considers exactly one transition.

Algorithm 7: `CheckNext`. `CheckNext` is mutually recursive with `Check`.

Require: ϕ : pCTL formula $\mathtt{P}_{\bowtie \mathtt{p}}[\mathtt{X}\ \phi_2]$
Return : $Sat_K(\phi)$: abstract states that satisfy ϕ

1 $\mathtt{S}_2 = \mathtt{Check}(\phi_2)$
2 $V_0^p := \{1.0 \leftarrow s | s \in \mathtt{S}_2\} \cup \{0 \leftarrow \emptyset\}$
3 $V_1^p := \mathtt{OneIteration}(V_0^p, \mathtt{X}\ \mathtt{S}_2)$
4 $Sat_K(\phi) := \emptyset$
5 **foreach** $p_i \leftarrow s_i \in V_1^p$ **do**
6 **if** $p_i \bowtie \mathtt{p}$ **then** add s_i to $Sat_K(\phi)$

5.2 Properties of PCTL-REBEL

PCTL-REBEL is a relational model checking algorithm that finds all states that satisfy a given pCTL formula. We discuss the properties of pCTL-REBEL.

Lifted. instead of operating at the ground level, pCTL-REBEL performs lifted inference as both the formula and the states are specified at an abstract level using relational representations. Using lifted inference allows pCTL-REBEL to exploit relational symme-

tries in the model and make abstraction of the domain, hence mitigate the state explosion problem. The lifted inference was discussed in detail in Sect. 4.

Sound for step-bounded pCTL formulae. PCTL-REBEL is sound for finite RMDPs (that have a finite domain) and any step-bounded pCTL formulae. PCTL-REBEL is not sound for indefinite-horizon formulae (e.g. `U`), just like many other value iteration algorithms. This is because pCTL-REBEL uses a naive termination criterion with some arbitrary convergence threshold ϵ (see `CheckUntil`, Algorithm 6). Although this naive termination criterion is not sound, it achieves precise approximation in practice. For further details, please refer to Haddad and Monmege (2014).

Complete. PCTL-REBEL is complete for finite RMDPs such that the state probability function V^p assigns a probability to all states. PCTL-REBEL captures the entire state space by using an ordered set of relational V^p-rules. Those states that are not captured by any other V^p-rules are guaranteed to be covered by the last V^p-rule $0 \leftarrow \emptyset$ (see Sect. 4.1).

6 Relational model checking for infinite MDPs

It is clear that when the domain is finite, the model checking problem is decidable as one can enumerates all states in the model, as discussed in Sect. 3.3. However, when the domain is infinite, the state space is typically infinite, making enumerating all states infeasible. In this section, we obtain decidability for a special class of infinite RMDPs. We will prove that under the *state-boundedness* condition (Belardinelli et al. 2011), a *finite abstraction* of such RMDPs can be constructed checked by pCTL-REBEL. The main idea is to generate a finite abstraction that captures all relevant information of the underlying infinite RMDP with respect to a pCTL formula. Accordingly, checking the abstraction is equivalent to checking the infinite RMDP. By checking the finite abstraction, the model checking problem becomes decidable. This section adopts the approach of Belardinelli et al. (2011) to construct finite abstractions of infinite RMDPs. Furthermore, we show that pCTL-REBEL can naturally handle such abstractions as they are structurally similar to RMDPs.

Section 6.1 defines the state-boundedness condition and under which, the finite abstraction of an infinite RMDP. Section 6.2 proves that checking a pCTL formula ϕ against the finite abstraction is equivalent to checking ϕ against the corresponding infinite RMDP. Section 6.3 discusses properties of pCTL-REBEL when handling such infinite RMDPs.

6.1 Abstract MDP: a finite abstraction of an infinite RMDP

Given an infinite RMDP, the relational model checking problem is generally undecidable due to the possibly infinite domain. To obtain decidability, this section constructs a finite abstraction of a given infinite RMDP, called an *abstract RMDP* (ARMDP). The purpose of an ARMDP is to use a finite model to capture all relevant information about a pCTL formula. An ARMDP must be constructed under the *state-boundedness* condition.

The state-boundedness condition states that any state concerns only a finite number of objects. For example, consider a blocks world that has infinitely many blocks and a table with a capacity b. An agent can take a block away or put a new block to the table, but no more than b blocks can be on the table at any moment. Hence, the state bound is b. Since any two states can describe totally different blocks, the model still contains *infinitely many* states. We say the blocks on the table are in the *active domain*. We now formally define active domain and state-boundedness.

Definition 6 A bounded state s_b is a finite subset of a ground state s in some RMDP, i.e. $s_b \subseteq s$. The active domain of s_b, denoted $adom(s_b)$, is the set of all domain objects in s_b.

An active domain $adom(s_b)$ is by definition finite as a bounded state s_b is finite. A bounded state is similar to a ground state but concerns only a finite number of objects. That is, all atoms in s_b are *true* and all the others are *false*.

Definition 7 Given an RMDP $K = \langle \Sigma, \Delta \rangle$, its underlying MDP $M = \langle S, A, T \rangle$, a state bound b, and a starting state s_0 such that $|adom(s_0)| \leq b$, an MDP $M_b = \langle S_b, A_b, T_b \rangle$ can be defined by including all states that are reachable from s_0 and contain at most b constants. Formally, $M_b = \langle S_b, A_b, T_b \rangle$ is defined as

$$
\begin{aligned}
S_b &:= \{s \mid s \in S, |adom(s)| \leq b\} \\
T_b &:= \{h \xleftarrow{p:a} b \in T | h \in S_b, b \in S_b\} \\
A_b &:= \{a | h \xleftarrow{p:a} b \in T_b\}
\end{aligned}
$$

If $b \in \mathbb{N}$, then M_b is called state-bounded or b-bounded.

A b-bounded MDP M_b is uniquely defined by an RMDP and a state bound. Roughly speaking, M_b a *sub-MDP* of M that concerns at most b objects in any state.

Example 8 Consider a blocks world $K = \langle \Sigma, \Delta \rangle$ with $\Sigma = \langle R, D \rangle$ where $R = \{\texttt{cl/1}, \texttt{on/2}\}$, $D = \{\texttt{bl}_\texttt{i} | i \in \mathbb{N}\}$ is infinite, and Δ contains the following rules (as in Fig. 1).

$$
\begin{aligned}
\delta_{\texttt{move}_1} &: \texttt{cl(A)},\texttt{cl(C)},\texttt{on(A, B)} \xleftarrow{\texttt{0.9:move(A,B,C)}} \texttt{cl(A)},\texttt{cl(B)},\texttt{on(A, C)} \\
\delta_{\texttt{move}_2} &: \texttt{cl(A)},\texttt{cl(B)},\texttt{on(A, C)} \xleftarrow{\texttt{0.1:move(A,B,C)}} \texttt{cl(A)},\texttt{cl(B)},\texttt{on(A, C)}
\end{aligned}
$$

Given a state bound b, the RMDP K defines a b-bounded MDP $M_b = \langle S_b, A_b, T_b \rangle$ such that each bounded state $s_b \in S_b$ contains at most b blocks. We will use this blocks world example throughout this section.

Having defined state-boundedness, let us now move on to constructing an abstract MDP (ARMDP). With respect to a pCTL formula, some domain objects in an MDP are irrelevant. For example, $\texttt{P}_{\geq 0.9}[\texttt{X cl(a)}]$ does not concern any particular blocks other than block a. By abstracting away irrelevant objects of a pCTL formula ϕ, we can construct an ARMDP M_ϕ that captures all necessary information to check ϕ. Concretely, to construct an ARMDP, all *ϕ-relevant* domain objects are preserved, and all other objects are abstracted using variables. A domain object is *ϕ-relevant* if and only if it is in ϕ or in a transition rule. Furthermore, if a formula ϕ concerns a finite number of objects, the corresponding ARMDP M_ϕ is finite. In this paper, we assume all formulae contain at most b objects where b is the state bound.

Definition 8 For a b-bounded MDP $M_b = \langle S_b, A_b, T_b \rangle$ of an RMDP $K = \langle \Sigma, \Delta \rangle$ and a relational pCTL formula ϕ, a b-bounded abstract RMDP $M_\phi = \langle S_\phi, \Sigma, \Delta, W \rangle$ is defined where

$$S_\phi = \{s_\phi | \exists s \in S_b.\ s \preceq_\theta s_\phi, \\ consts(s_\phi) \subseteq consts(\phi) \cup consts(\Delta), \\ vars(s_\phi) \subseteq W\}$$

and W is a set of $b - |consts(\phi) \cup consts(\Delta)|$ distinct variables. All terms in $s_\phi \in S_\phi$ are from a finite set of terms

$$terms(s_\phi) := W \cup consts(\phi) \cup consts(\Delta)$$

It is assumed that $vars(\phi) \in W$. It is assumed that states in S_ϕ are not syntactic variants, that is, they are not a variable renaming of one another.

An ARMDP is finite as only a finite number of terms are allowed in the state description. An ARMDP is b-bounded as each state $s_\phi \in S_\phi$ contains at most b terms, i.e. $|terms(s_\phi)| \leq b$. Since an ARMDP state is b-bounded, it has bounded branching behavior such that the number of available actions in any state is finite. The finite state space of ARMDPs is crucial to obtaining decidability.

An ARMDP has a similar structure as an RMDP but has a more abstract state space. That is, unlike RMDPs, the state space of an ARMDP is not connected to an explicit domain as any ARMDP state concerns only ϕ-relevant constants and variables. By replacing the variables by domain constants, an ARMDP state can enumerate infinitely many ground states in the underlying RMDP.

Example 9 (Cont. Example 8) A b-bounded MDP $M_b = \langle S_b, A_b, T_b \rangle$ and the formula $\phi = \texttt{cl(a)}$ defines an ARMDP $M_\phi = \langle S_\phi, \Sigma, \Delta, W \rangle$. A state $s_b \in S_b$ concerns at most b terms, namely $\{\texttt{Bl}_1, \texttt{Bl}_2, ..., \texttt{Bl}_{\texttt{b}-1}, \texttt{a}\}$. The abstract states $s_1 = \{\texttt{cl(a)}\} \in S_\phi$ and $s_2 = \{\texttt{cl}(\texttt{Bl}_1)\} \in S_\phi$ represent the set of all infinitely many ground states that have at least one clear block. These ground states can be enumerated by assigning domain objects to $\texttt{Bl}_1$.

This section has shown that given a state-bounded MDP and a pCTL formula, an finite abstraction can be constructed. Such abstraction is called an ARMDP. An ARMDP is similar to an RMDP but is state-bounded, finite, and captures only ϕ-relevant information. The next section will show that checking an ARMDP is equivalent to checking the underlying model.

6.2 Decidable model checking for ARMDPs

This section proves the decidability of the model checking problem for a special class of infinite RMDPs, namely, the ARMDP, defined in Sect. 6.1. Decidability is obtained by proving that checking the ARMDP yields equivalent results as checking the underlying infinite, state-bounded MDP, namely, Theorem 1.

Theorem 1 *For a b-bounded MDP M_b and its corresponding ARMDP M_ϕ based on a pCTL sentence ϕ, checking ϕ against M_ϕ is equivalent to checking ϕ against M_b, formally,*

$$M_b \vDash \phi \Leftrightarrow M_\phi \vDash \phi$$

Theorem 1 is proven by using probabilistic bisimulation. Probabilistic bisimulation is a standard model checking technique that compares two probabilistic transition systems (Baier and Katoen 2008). Given a pCTL formula ϕ, if two states from different transition systems are *probabilistic bisimilar*, then their behaviors are indistinguishable. Hence, checking ϕ in either system yields identical results. Theorem 1 extends the theorem by Belardinelli et al. (2011)[Theorem 2] for the probabilistic setting.[5]

We prove Theorem 1 in two steps. First, we define indistinguishable states (Definition 9) and probabilistic bisimulation (Definition 10). Second, we show that an ARMDP and its underlying state-bounded MDP actually define a probabilistic bisimulation (Propositions 1 and 2).

Definition 9 Given the b-bounded MDP $M_b = \langle S_b, A_b, T_b \rangle$ of a RMDP $K = \langle \Sigma, \Delta \rangle$ and a pCTL sentence ϕ, consider two ground states s_1 and s_2 in S_b such that $C_1 = consts(s_1) \subset D$ and $C_2 = consts(s_2) \subset D$. Let C be the set of all ϕ-relevant domain objects $C := consts(\phi) \cup consts(\Delta) \subseteq C_1 \cap C_2$. The two states s_1 and s_2 are called *indistinguishable*[6] under C, if and only if s_1 and s_2 are renamings of one another under a bijection $f : C_1 \backslash C \mapsto C_2 \backslash C$. The bijection f renames ϕ-irrelevant domain objects. Formally,

$$s_1 \sim_C s_2 \iff f(s_1) = s_2$$

We abuse the notation and let $f(s_1)$ be the state obtained by renaming constants in s_1.

Definition 9 has defined indistinguishable states under a pCTL formula ϕ. Indistinguishable states are essentially renamings of one another and share the same properties with respect to a formula ϕ, hence, they can be represented by one abstract state in an ARMDP. That is, an ARMDP state represents infinitely many indistinguishable states in the underlying MDP.

Example 10 (Cont. Example 9) Consider a state-bounded MDP $M_b = \langle S_b, A_b, T_b \rangle$ with a state bound $b = 2$, a pCTL formula $\phi = \texttt{cl(a)}$ and two ground states $s_3 = \{\texttt{cl(b)}, \texttt{on(b,a)})\} \in S_b$ and $s_4 = \{\texttt{cl(c)}, \texttt{on(c,a)})\} \in S_b$. States s_3 and s_4 are variable renamings of each other thus indistinguishable under ϕ. They both express *some block is on block* a and can be represented by the abstract state $s_5 = \{\texttt{cl(Bl}_1\texttt{)}, \texttt{on(Bl}_1\texttt{,a)}\}$.

Now we must define *probabilistic bisimulation*.

Definition 10 [cf. Baier and Katoen (2008)] Consider a b-bounded MDP $M_b = \langle S_b, A_b, T_b \rangle$, a pCTL formula ϕ and the finite set C of ϕ-relevant constants, $\mathcal{R} \subseteq S_b \times S_b$ is a *probabilistic bisimulation* if for any pair of states $\langle s_1, s_2 \rangle \in \mathcal{R}$:

1. s_1 and s_2 are indistinguishable under C, i.e. $s_1 \sim_C s_2$
2. s_1 and s_2 have the identical probabilities of going to any other distinguishable states, i.e. $\forall a \in A : T(s_1, a)(s) = T(s_2, a)(s)$ for each $s \in S/\mathcal{R}$

[5] More specifically, as compared to their work, this paper uses RMDPs instead of artifact systems, MDPs instead of Kripke structures, ground states instead of databases, and probabilistic bisimulation instead of bisimulation.

[6] Indistinguishable states are also commonly called isomorphic states.

States s_1 and s_2 are called probabilistic bisimilar.

To prove Theorem 1, we must prove that M_ϕ and M_b indeed form a probabilistic bisimulation. Propositions 1 and 2 will together prove Theorem 1, i.e. checking an ARMDP is equivalent to checking its underlying state-bounded MDP.

Proposition 1 *For an ARMDP $M_\phi = \langle S_\phi, \Sigma, \Delta, W\rangle$ based on a b-bounded MDP $M_b = \langle S_b, A_b, T_b\rangle$ and a pCTL sentence ϕ, if two states are indistinguishable, then they either both can fire a transition rule or both cannot fire a transition rule. Formally, for any transition rule of the form*

$$H_i \xleftarrow{p_i:\alpha} B$$

if $s_1 \sim_C s_2$, then

$$s_1 \preceq_{\theta_1} B \iff s_2 \preceq_{\theta_2} B$$

for two indistinguishable substitutions θ_1 and θ_2.

Proposition 2 *For an ARMDP $M_\phi = \langle S_\phi, \Sigma, \Delta, W\rangle$ based on a b-bounded MDP $M_b = \langle S_b, A_b, T_b\rangle$ and a pCTL sentence ϕ, if two states are indistinguishable, then they have the identical probabilities of going to any other distinguishable states. Formally, for any transition rule $\delta \in \Delta$ of the form*

$$\{H_1 \xleftarrow{p_1:\alpha} B, ..., H_n \xleftarrow{p_n:\alpha} B\}$$

if $s_1 \sim_C s_2$, then they define two sets of ground transition rules $T(s_1, \alpha\theta_1)$ and $T(s_2, \alpha\theta_2)$ that share the same transition probabilities.

$$\begin{aligned} T(s_1, \alpha\theta_1) := \{h_{1,i} \xleftarrow{p_i:\alpha\theta_1} s_1 | H_i \xleftarrow{p:\alpha} B \in \delta, s_1 \preceq_{\theta_1} B, \\ h_{1,i} = (s_1 \backslash B\theta_1) \cup H_i\theta_1\} \\ T(s_2, \alpha\theta_2) := \{h_{2,i} \xleftarrow{p_i:\alpha\theta_2} s_2 | H_i \xleftarrow{p:\alpha} B \in \delta, s_2 \preceq_{\theta_2} B, \\ h_{2,i} = (s_2 \backslash B\theta_2) \cup H_i\theta_2\} \end{aligned}$$

Propositions 1 and 2 have shown that the indistinguishable relation $\sim_C$ between an ARMDP and its underlying state-bounded MDP is a probabilistic bisimulation. $\sim_C$ is an equivalence relation such that two probabilistic bisimilar states exhibit identical behavior. In other words, the ARMDP and the underlying MDP denote a mutual, step-wise simulation of indistinguishable states. With a similar reasoning as in Belardinelli et al. (2011), we conclude that the model checking problem for infinite, state-bounded MDPs is decidable as it can be done on its corresponding finite ARMDP.

6.3 PCTL-REBEL for infinite MDPs

PCTL-REBEL can handle an infinite, state-bounded RMDP by reasoning about its finite abstraction. More formally, given an infinite RMDP with a step bound and a pCTL formula, a finite ARMDP can be constructed and naturally checked by pCTL-REBEL. In fact, as an ARMDP is just like an RMDP that concerns at most b objects in a state (see Sect. 6.1), pCTL-REBEL requires only one adaption to restrict the state size. That is, all

states that have more than b objects must be eliminated. The number of objects in a state bound is simply obtained by counting all variables and constants as OI-subsumption is imposed. The adaption of maintaining the state bound is implemented in `Regression` (Algorithm 2).

PCTL-REBEL for an infinite, state-bounded MDP is lifted and complete, just as like pCTL-REBEL for a finite RMDP (see Sect. 5.2). We discuss properties of pCTL-REBEL for state-bounded MDPs in detail.

Lifted. PCTL-REBEL checks a state-bounded, infinite RMDP that has a finite abstraction (namely, an ARMDP) at a lifted level. Specifically, pCTL-REBEL operates on the ARMDP that is bisimilar to the underlying ground MDP. Checking an ARMDP is the same as checking an RMDP, except that the state bound must be maintained.
Sound for step-bounded pCTL formulae. PCTL-REBEL is sound for step-bounded pCTL formulae in an ARMDP. This is a consequence of Theorem 1 that shows checking an ARMDP yields exactly the same results as checking its underlying MDP. PCTL-REBEL is not sound for indefinite-horizon formulae but it achieves precise approximation in practice.
Complete. PCTL-REBEL is complete for checking ARMDPs as all states are assigned a probability. PCTL-REBEL captures the entire state space by using an ordered set of relational V^p-rules.

7 Experiments

We aim to answer the following questions in this section. Q1 and Q2 focus on the benefits of pCTL-REBEL, and Q3-Q5 focus on the limitations of pCTL-REBEL.

Q1 What formulae can pCTL-REBEL check in practice?
Q2 How does pCTL-REBEL compare with state-of-the-art model checkers?
Q3 How well does pCTL-REBEL handle indefinite-horizon formulae?
Q4 How well does pCTL-REBEL handle a complex relational transition function?
Q5 What are the computational costs of different pCTL operators?

We implemented and validated an unoptimized pCTL-REBEL research prototype using SWI-Prolog 8.0.2, with the *constraint handling rules* library. Experiments were run on a 2.4 GHz Intel i5 processor. We use the blocks world dataset[7] and the box world dataset.[8] We compare pCTL-REBEL with the state-of-the-art model checkers PRISM (Kwiatkowska et al. 2011) and STORM (Dehnert et al. 2017).[9] We set a time-out of 1800 s for all model checkers. PCTL-REBEL's state bound is sometimes referred as *number of blocks/cities* so as to provide a direct comparison with PRISM and STORM that operate on ground models.

The blocks world and the box world datasets originate from IPPC-2008[10] and are originally specified in the ppddl language. STORM and PRISM operate on the converted

[7] https://qcomp.org/benchmarks/#blocksworld.
[8] https://qcomp.org/benchmarks/#boxworld.
[9] The experiments will be made public once the paper is accepted.
[10] http://ippc-2008.loria.fr/wiki/index.php/Results.html#Fully_Observable_Non-Deterministic_.28FOND.29_track_2.

Table 3 PCTL-REBEL can perform a range of pCTL properties in the *infinite* blocks and box worlds under a time-out of 1800 seconds

Domain	Property	Formula	Runtime (s)
Blocks	Bounded reachability with det. actions	$\mathtt{P}_{\geq 0.5}[\mathtt{F}^{\leq 10}\mathtt{on(a,b)}]$	11
Blocks	Bounded reachability	$\mathtt{P}_{\geq 0.5}[\mathtt{F}^{\leq 10}\mathtt{on(a,b)}]$	226
Blocks	Bounded constrained reachability	$\mathtt{P}_{\geq 0.5}[\mathtt{on(c,d)U}^{\leq 5}\mathtt{on(a,b)}]$	7.1
Blocks	Nested	$\phi_{\mathtt{nested}}(3,1)$	1554
Box	Unbounded reachability	$\mathtt{P}_{\geq 0.5}[\mathtt{F\ bin(b1,paris)}]$	0.7
Box	Bounded constrained reachability	$\mathtt{P}_{\geq 0.5}[\mathtt{tin(t1,paris)U}^{\leq 5}\mathtt{bin(b1,paris)}]$	0.365

Kersting et al. (2004) reported a runtime of roughly 10 min for this task

prism-format files,[11] and pCTL-REBEL operates on the converted prolog-format files.[12] We discard the reward structure in the model. The blocks world dataset is simplified to have the relations `on/2`, `clear/1` and the action `move/3` that has a success probability of 0.9. The box world dataset has the relations `bin/2`, `on/2`, `tin/2`, `can−drive/2` and the actions `drive/3`, `load/2`, `unload/2`. In the box world, `bin(B, C)` expresses a box is in a city, `on(B, T)` expresses a box is on a truck, and `tin(T, C)` expresses a truck is in a city. An atom `can−drive(C1, C2)` expresses a road that directly connects `C1` and `C2`. A box can be loaded on (`load(B, T)`) or unloaded from a truck (`unload(B, T)`). A truck can travel from a city to another (`drive(T, C1, C2)`). The `load/2` and `unload/2` actions succeed with probability 0.9. The `drive/2` action succeeds with probability 0.8.

7.1 Q1: What formulae can pCTL-REBEL check in practice?

Although pCTL-REBEL can check any formulae of the pCTL language as discussed in Sect. 3.2, in practice, some formulae are more costly than others. This experiment aims at evaluating the computational costs of different pCTL formulae. Table 3 includes formulae ranging from classic planning properties, i.e. reachability properties (Kersting et al. 2004; Boutilier et al. 2001; Sanner and Boutilier 2009), to more complex, nested pCTL properties, e.g. the following $\phi_{\mathtt{nested}}(i,j)$ formula. Recall that $\phi_{\mathtt{nested}}(4,2)$ was discussed in Example 7, and its parse tree is in Fig. 4. Table 3 also includes a formula that has an indefinite horizon, namely Property 5.

$$\phi_{\mathtt{nested}}(i,j) = \mathtt{P}_{\geq 0.5}[\mathtt{cl(a)U}^{\leq \mathtt{i}}(\mathtt{on(a,b)} \wedge \mathtt{P}_{\geq 0.8}[\mathtt{P}_{\geq 0.9}[\mathtt{X\ cl(e)}]\mathtt{U}^{\leq \mathtt{j}}\mathtt{on(c,d)}])]$$

We now discuss Table 3 in detail. First, all formulae in Table 3 are checked against models that have *infinitely many* objects, which is possible only because pCTL-REBEL uses lifted inference. Second, pCTL-REBEL can handle classic planning tasks by formulating them as reachability formulae (Property 1, 2, 5, Table 3) or constrained reachability formulae (Property 3,6, Table 3). For example, Fig. 5 visualizes the solution to Property 2 in Table 3. Third, pCTL-REBEL can check, in an infinite model, formulae that have an infinite horizon (Property 5, Table 3). This property cannot be evaluated by an explicit-state model

[11] The ppddl models are first converted to the jani format by the ppddl2jani tool (attached in the datasets), then converted to the prism format by ePMC (Hahn et al. 2014).

[12] The ppddl models are translated to prolog models as PCTL-REBEL is implemented in prolog.

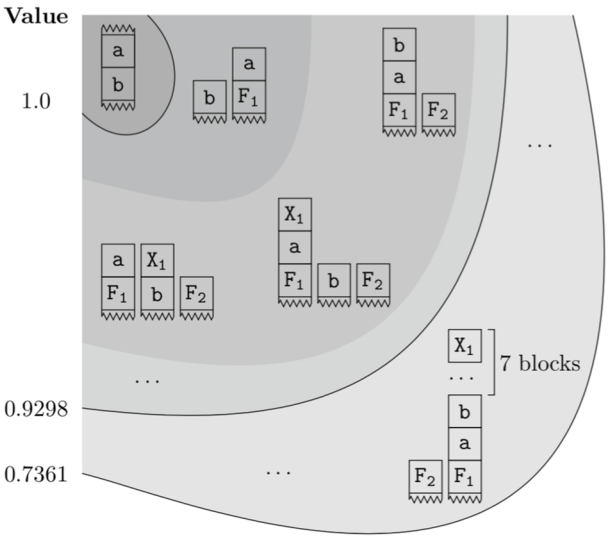

Fig. 5 The abstracts states that satisfy $\mathrm{P}_{\geq 0.5}[\mathrm{F}^{\leq 10}\mathtt{on(a,b)}]$. Any state subsumed by $\{\mathtt{on(a,b)}\}$ is a goal state thus has a value 1. The states that are further away from the upper-left corner require more successful actions to reach $\{\mathtt{on(a,b)}\}$ thus have a smaller probability

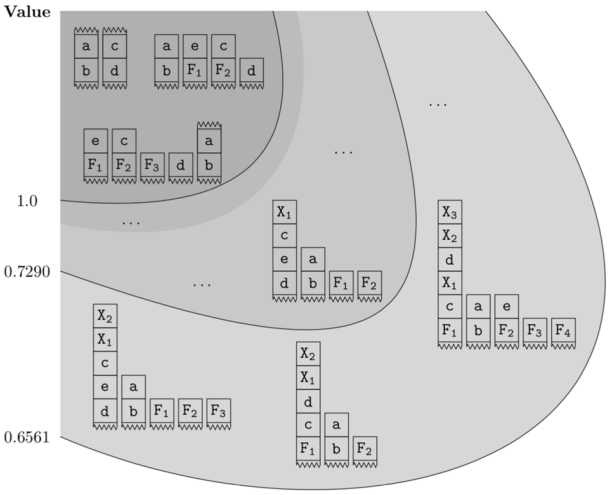

Fig. 6 Similar to Fig. 5, this illustrates the abstract states that satisfy $\phi_{\mathtt{nested}}(4,2)$ in the blocks world

checker as state space is infinite. Finally, pCTL-REBEL can handle nested formulae such as $\phi_{\mathtt{nested}}(i,j)$ (Property 4, Table 3). Under a time-out of 1800 seconds, pCTL-REBEL can compute $\phi_{\mathtt{nested}}(3,1)$, which is cheaper than computing $\phi_{\mathtt{nested}}(4,2)$. Nonetheless, Fig. 6 visualizes the results of $\phi_{\mathtt{nested}}(4,2)$ by relaxing the time-out. More discussion about nested formulae will be in Q5.

7.2 Q2: How does pCTL-REBEL compare with state-of-the-art model checkers?

State-of-the-art model checkers STORM and PRISM do not operate on an RMDP as in Q1. They require the RMDP to be grounded, which leads to a state explosion. This experiment

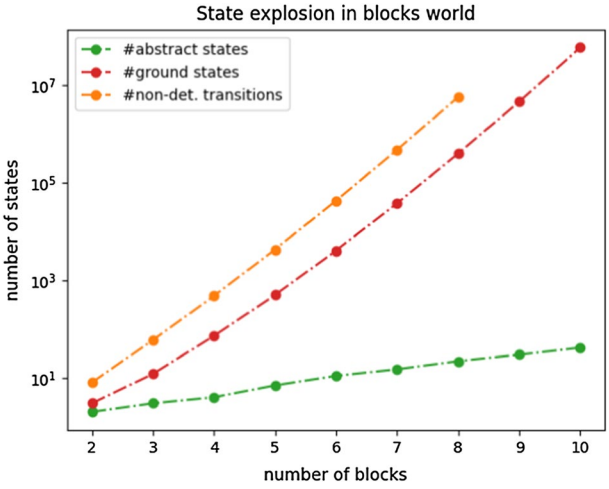

Fig. 7 The blocks world stats. When the number of blocks grows, the number of ground states and the number of nondeterministic transitions explode more than the minimum number of abstract states

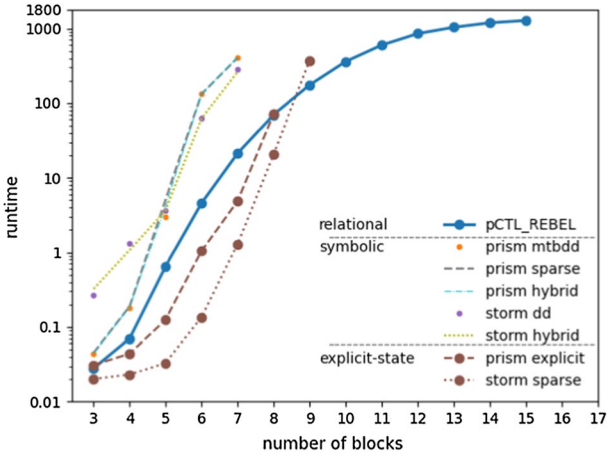

Fig. 8 The property $P_{\geq 0.5}[F^{\leq 10}\texttt{on(a, b)}]$ of is checked by three model checkers. With a time-out of 1800 seconds, PRISM can handle at most 8 blocks with the explicit engine, STORM can handle at most 9 blocks with the sparse engine, and pCTL-REBEL can handle 15 blocks

evaluates how well STORM and PRISM handle such state explosions in practice. In particular, we show that pCTL-REBEL, by lifting, performs better in mitigating state explosions in relational domains. While Q1 assumes that the given RMDP has an infinitely large domain, in order to compare pCTL-REBEL with STORM and PRISM, this section considers finite domains such that explicit-state models can be generated. Furthermore, we increment the domain size by 1 to track the trends in computation time for all model checkers.

The state explosion problem is illustrated in Fig. 7, showing the number of ground states (Slaney and Thiébaux 2001) and the corresponding ground transitions[13] grow much faster than the minimum number of abstract states.[14] Even though all numbers grow

[13] The number of ground transitions is obtained from the STORM model checker.

[14] The minimum number of abstract states is the number of integer partitions that capture all relational structures. For example, for three blocks, there are three abstract states $\{\texttt{cl(A)}, \texttt{cl(B)}, \texttt{cl(C)}\}$, $\{\texttt{cl(A)}, \texttt{cl(B)}, \texttt{on(B, C)}\}$ and $\{\texttt{cl(A)}, \texttt{on(A, B)}, \texttt{on(B, C)}\}$.

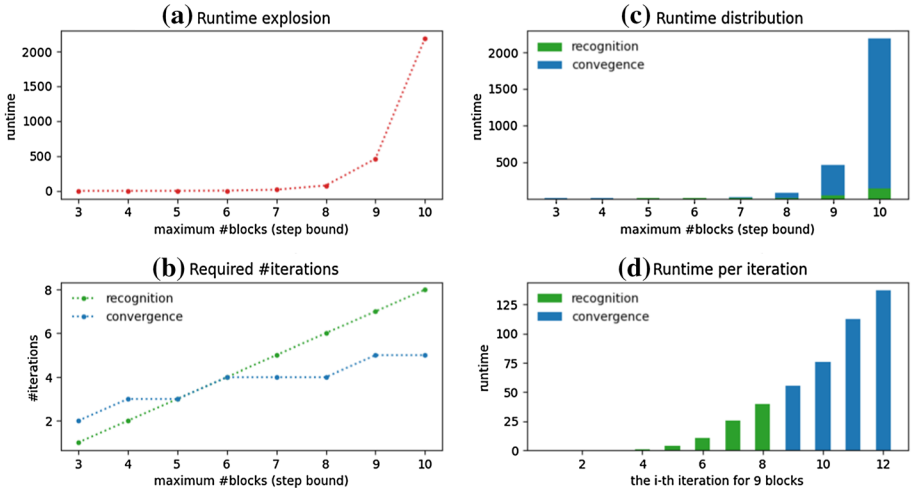

Fig. 9 The results obtained from checking the reachability property $\mathtt{P}_{\geq 0.5}[\mathtt{F}\ \mathtt{on(a,b)}]$ in the blocks world. **a** The runtime increases exponentially to the number of required iterations. Most of the runtime is taken by the value convergence stage. **b** The later iterations are more expensive than the earlier iterations

exponentially, the minimum number of abstract states is the most resistant to the growth of the domain size.

We compare the scalability of the three model checkers (including 4 PRISM engines and 3 STORM engines) in the blocks world. We could essentially select any pCTL property for the comparison. Nonetheless, for simplicity, we use a simple reachability property $\mathtt{P}_{\geq 0.5}[\mathtt{F}^{\leq 10}\mathtt{on(a,b)}]$ (Property 2, Table 3). We measure how the runtime grows with respect to the domain size, shown in Fig. 8. The results show that pCTL-REBEL is more scalable than any PRISM or STORM engine in relational domains. Under a time-out of 1800 seconds, pCTL-REBEL can handle 15 blocks (6.6e13 ground states) whereas PRISM can handle at most 8 blocks (4.0e5 ground states) and STORM can handle at most 9 blocks (4.6e6 ground states).

We analyze Fig. 8 to compare lifting with other optimization techniques. Observe that Fig. 8 categorizes all engines in three groups: (1) *relational* model checking: pCTL-REBEL, (2) *symbolic* model checking: PRISM's mtbdd, sparse, hybrid engines and STORM's dd, hybrid engines, and (3) *explicit-state* model checking: PRISM's explicit engine and STORM's sparse engine. Compared to pCTL-REBEL, not only the explicit-state approaches but also the symbolic approaches are much less resilient to the growth of the domain size. Clearly, although symbolic model checking methods are known to be able to handle large domains, their ability of handling state explosions in relational domains is limited. Furthermore, unlike other engines, the runtime of pCTL-REBEL saturates when the domain grows to a certain extent. This phenomenon is a consequence of lifted inference, reflecting that the number of relational structures will eventually saturate.

7.3 Q3: How well does pCTL-REBEL handle indefinite-horizon formulae?

PCTL-REBEL suffers from runtime explosions when checking a formula that has an indefinite horizon. In specific, the runtime grows exponentially with respect to the horizon. This

experiment aims at analyzing the cause of the runtime explosion. We use the reachability property $\mathtt{P}_{\geq 0.5}[\mathtt{F\ on(a,b)}]$ that has an indefinite horizon.

To analyze the cause of the runtime explosion, we split the *iterations* of checking an indefinite-horizon formula into two stages: (1) the *state recognition stage* and (2) the *value convergence stage*. In the state recognition stage, pCTL-REBEL identifies new satisfying states in each iteration. This stage ends when no more new satisfying states are found. In the value convergence stage, pCTL-REBEL does not discover new states, and only updates probabilities of all identified states until convergence. Notice that these two stages are just for analyzing purposes as pCTL-REBEL conducts exactly the same computation in all iterations in both stages.

We show the runtime growth of $\mathtt{P}_{\geq 0.5}[\mathtt{F\ on(a,b)}]$ in Fig. 9 with respect to the domain size. Fig. 9a shows the runtime explodes with respect to *the number of required iterations*, and illustrates that most of the runtime is taken by the value convergence stage. Moreover, Fig. 9b shows that within one checking process, later iterations are more expensive than earlier iterations. This means that although the job of the value convergence state is seemingly easier than the state recognition stage, in practice, it takes more time. This extra computation time comes from pCTL-REBEL's attempts at identifying new states in the value convergence stage.

In summary, pCTL-REBEL suffers from a runtime explosion when handling indefinite formulae. The main cause is the redundant computation conducted in the value convergence stage. That is, pCTL-REBEL performs the same computation without being aware of the two-stage nature of the model checking process. This problem can be mitigated by optimizing the algorithm so as to reduce the time taken by the second stage. The optimization should benefit all indefinite-horizon formulae.

7.4 Q4: How well does pCTL-REBEL handle a complex relational transition function?

The use of abstract actions is key to the scalability of RMDPs as it allows for reasoning about a large group of ground actions as a whole. Roughly speaking, the more ground actions an abstract action captures, the more scalable pCTL-REBEL can be. For example, the blocks world domain has a simple `move` abstract action (see Fig. 1) that captures countless ground actions. Therefore, pCTL-REBEL is very scalable to the growth of the domain size. However, it is common to have constraints on the action, which significantly harms scalability. An action constraint harms scalability by prohibiting certain substitutions for an abstract action. In other words, it forces the abstract action to depend on the true identity of some objects. Action constraints break relational structures and result in non-symmetric transition functions. The resulting transition function consists of more rules and cannot be represented as compactly anymore. We call this constrained transition functions *relationally* complex.

The following example illustrates action constraints. Consider a box world domain where the action `drive(T,C1,C2)` is constrained by the underlying road map below.[15] A truck `T` can drive from `C1` to `C2` only if `can_drive(C1,C2)` exists.

[15] The road networks are automatically generated by the box world dataset.

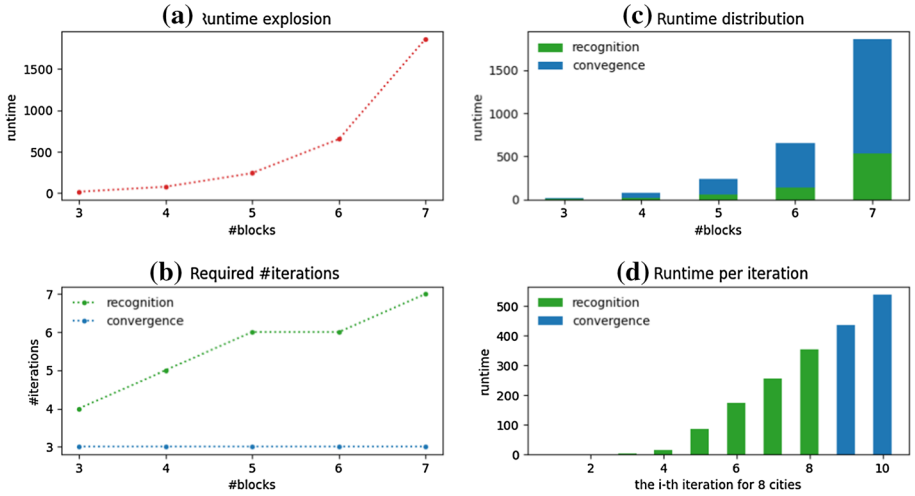

Fig. 10 The results obtained from checking the reachability property $\mathrm{P}_{\geq 0.5}[\mathrm{F}\ \mathtt{bin(b1, city0)}]$ in the box world. The runtime increases exponentially when the transition structure becomes more complex

```
can-drive(city0,city2). can-drive(city2,city1). can-drive(city5,city4).
can-drive(city0,city1). can-drive(city2,city6). can-drive(city5,city1).
can-drive(city0,city6). can-drive(city3,city1). can-drive(city5,city3).
can-drive(city1,city0). can-drive(city3,city6). can-drive(city6,city0).
can-drive(city1,city6). can-drive(city3,city4). can-drive(city6,city1).
can-drive(city1,city3). can-drive(city3,city5). can-drive(city6,city2).
can-drive(city1,city4). can-drive(city4,city1). can-drive(city6,city3).
can-drive(city1,city2). can-drive(city4,city3). can-drive(city6,city4).
can-drive(city1,city5). can-drive(city4,city6).
can-drive(city2,city0). can-drive(city4,city5).
```

This experiment examines how well pCTL-REBEL handles complex relational transition functions. We use the reachability property $\phi = \mathrm{P}_{\geq 0.5}[\mathrm{F}\ \mathtt{bin(b1, city0)}]$ and automatically generated road maps. Figure 10 shows the results that pCTL-REBEL can handle a road map containing at most 8 cities under a time-out of 1800 seconds. The runtime increases exponentially with respect to the size of the road network. Recall that without a road network, it takes only 0.7 seconds to check ϕ (see Property 5, Table 3). Since the road network makes the model dynamics more complex, more transition rules are required, resulting in runtime explosions. In short, pCTL-REBEL works the best when the dynamics of the objects is simple, e.g. in the blocks world or in the box world without a road map.

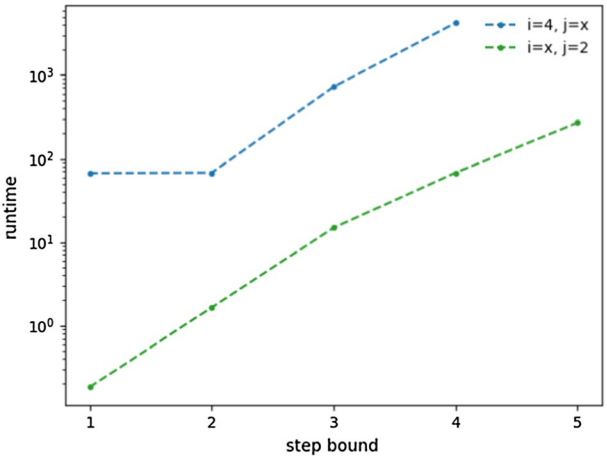

Fig. 11 The runtime of $\phi_{\texttt{nested}}(i,j)$ for different values of i and j. The argument i is the step bound of the outer U operator, and j is the step bound of the inner U operator. The runtime grows exponentially when either step bound increases. However, the runtime is more sensitive to the inner step bound j

7.5 Q5: What are the computational costs of different pCTL operators?

It is crucial to know what kind of formulae pCTL-REBEL can handle efficiently when designing properties for a model. This experiment aims at giving insight into how different factors affect pCTL-REBEL's performance.

Some path operators are more costly than others. e.g. the X operator is the cheapest since it requires only one iteration. The reachability operator F is generally more expensive than the constrained reachability operator U since U is a stricter operator that prunes out more states. For example, checking $\texttt{P}_{\leq 0.9}[\texttt{F}^{\leq 7}\ \texttt{on(a, b)}]$ (18.494 seconds) is more costly than checking $\texttt{P}_{\leq 0.9}[\texttt{on(c, d)}\ \texttt{U}^{\leq 7}\ \texttt{on(a, b)}]$ (0.733 seconds). A nested formula is more costly than a flat formula since it requires several recursive checking processes.

The formula structure also influences runtime. Specifically, changing an inner step bound has a larger impact than changing an outer step bound. Taking the formula $\phi_{\texttt{nested}}(i,j)$ (defined in Q1) for example, the parameters i and j are both step bounds for U formulae, but i is attached to an outer subformula and j is attached to the inner subformula. Hence, the total runtime is more sensitive to the change in j than to the change in i, as shown in Fig. 11. The cause of this difference is that the inner formulae are computed before the outer formulae. By changing an inner step bound, the number of satisfying states for the inner subformula changes, which directly influence the overhead of computing the outer subformulae.

8 Related work

We have introduced pCTL-REBEL, a relational model checking technique that reasons at the relational level and mitigates the state explosion problem in relational MDPs. PCTL-REBEL extends existing frameworks. First, pCTL-REBEL extends pCTL model checking with relational representations. As far as the authors know, pCTL-REBEL is the first relational technique for pCTL model checking. Second, pCTL-REBEL extends the decidability results of infinite, state-bounded MDPs to the probabilistic setting.

To mitigate the state explosion problem in model checking, several *grouping* techniques have been proposed. Different from our work, the following methods exploit various types of symmetries other than relational structures. *Symbolic model checking* (McMillan 1993) is a form of model checking that symbolically represents the state space and the set of states in which a formula holds. Symbolic model checking is often implemented using BDDs that are Boolean formulae in canonical form, in which isomorphic subformulae are shared, to allow efficient operations on the sets of states that they represent. Symbolic model checking has been first used for model checking of plain non-probabilistic finite state systems (McMillan 1993), then later adapted to the probabilistic case in Forejt et al. (2011). Contrary to our algorithm, these symbolic methods only apply to finite state MDPs. Another method to mitigate the state explosion problem is *abstraction-refinement*, which successively generates abstractions by partitioning the state space into *regions*. Therefore, the model checking algorithms handle regions instead of individual states (de Alfaro and Roy 2007; Roy et al. 2008). These techniques focus on grouping states into regions which have similar local properties but such regions are usually not exact as the states in a region can have different dynamics and future evaluations. Similarly, Marthi (2007) uses *abstract MDP* to aggregate not only states but also actions by exploiting the temporal dependency of actions. In addition, *game-based abstraction* techniques construct abstractions of MDPs by merging concrete transitions into abstract transitions (Kattenbelt et al. 2008). Different from our work, for the abstractions, they calculate approximate upper and lower bounds instead of exact values.

Many studies integrate model checking techniques into other AI fields. In planning, authors compute the optimal policy at the first-order level using value iteration (Boutilier et al. 2001; Sanner and Boutilier 2009; Kersting et al. 2004), which can be seen as a probabilistic reachability task, namely a special case of the general probabilistic model checking problem. PCTL-REBEL generalizes this special case with a temporal logic thus fits in the Planning as Model Checking paradigm (Giunchiglia and Traverso 2000). In robotics, model checking techniques are used for online motion planning (Lahijanian et al. 2012; Maly et al. 2013; He et al. 2015) where the states are labeled with propositions. These propositions result in a multidimensional state space that grows exponentially with the domain size (He et al. 2015), which is the exact problem that this paper tackles. Generally, our work can be applied a wide range of frameworks that have a large model, particularly in relational domains.

There is a fruitful literature on theoretical results of infinite systems. Particularly, the state-boundedness condition for decidable verification has been defined in first-order mu-calculus (Bagheri Hariri et al. 2013; Calvanese et al. 2018; De Giacomo et al. 2012; De Giacomo et al. 2015) and first-order CTL (Belardinelli et al. 2011, 2012). However, these studies focus on the non-probabilistic setting. Our work contributes to extending the theoretical results to stochastic models by introducing the abstract relational MDP. A limitation of the relational pCTL language is that it does not support quantification across conjunctions, which could be investigated as described in the work of Belardinelli et al. (2013).

9 Towards safe reinforcement learning

Recently, a converging interest has emerged about the pursuit of general dynamic systems that can autonomously *reason* and *learn* by interacting with the environment (Giacomo 2019). Since such systems require the ability of reasoning about

first-order state representations and safety (Amodei et al. 2016), pCTL-REBEL fits in as a first-order model checking technique that is also particularly relevant in *safe reinforcement learning*.

Safe reinforcement learning is the process of learning a policy that maximizes the expected reward in domains where safety constraints must be respected. In particular, *safe exploration*, i.e. guaranteeing an agent's safety in the exploration phase, is a non-trivial problem. Commonly used exploration strategies such as ϵ-greedy and softmax sometimes select a *random* action, which can result in catastrophic situations (Amodei et al. 2016). Existing research recognizes the importance of providing safety guarantees in reinforcement learning (Garcia and Fernández 2015; Pecka and Svoboda 2014; Lahijanian et al. 2012; Teichteil-Königsbuch 2012; Sprauel et al. 2014; Giunchiglia and Traverso 2000; Mason et al. 2018; Hasanbeig et al. 2019; Alshiekh et al. 2018; Jansen et al. 2020). However, most studies are based on techniques that require explicit state exploration whereas only few studies touch on safe exploration in relational domains (Driessens and Džeroski 2004; Martínez et al. 2017) Furthermore, several studies have investigated augmenting reinforcement learning with model checking techniques, including preventing the agent from taking risky actions (Alshiekh et al. 2018; Jansen et al. 2020; Fulton and Platzer 2018), synthesizing an initial safe *partial* policy for learning (Leonetti et al. 2012), learning a policy that maximizes the probability of satisfying a temporal formula (Hasanbeig et al. 2019), and shaping the reward function based on a temporal formula (De Giacomo et al. 2019). These approaches can be augmented with our work for scalability in large relational domains.

PCTL-REBEL can be used as a safe model-based reinforcement learning algorithm as it performs value iteration in a setting where goal states and safety constraints are formulated as pCTL formulae. More precisely, it derives the states in an RMDP that satisfy a given (relational) pCTL formula that encodes goal states and safety constraints. Indeed, pCTL-REBEL tackles a special safe reinforcement learning task where the reward is 1 for goal states and 0 for all other states, and the discount factor is 1 (Kersting et al. 2004; Yoon et al. 2012; Otterlo 2004). Formally, pCTL-REBEL in this paper

$$V_0^p(s) = \begin{cases} 0 & ,\forall s \notin G \\ 1 & ,\forall s \in G \end{cases}$$
$$V_{t+1}^p(s) = \max_a \sum_{s'} T(s,a,s') V_t^p(s')$$

is a special case of the standard Bellman operator

$$V_0^p(s) = \begin{cases} 0 & ,\forall s \notin G \\ r & ,\forall s \in G \end{cases}$$
$$V_{t+1}^p(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_t^p(s')]$$

where r is a reward value in $\mathbb{R}$.

It is an interesting avenue for further research to investigate representing general reward functions within this framework. Furthermore, pCTL-REBEL can be naturally combined with relational reinforcement learning (Džeroski et al. 2001) to achieve safe relational reinforcement learning as both frameworks use relational representations and work at an abstract level rather than at the ground level.

10 Conclusions

We have introduced a framework for lifted model checking in relational domains. To this aim, relational Markov Decision Processes have been integrated with model checking principles for pCTL. The result is a very expressive framework for model checking in probabilistic planning domains that are relational, that is, involve objects as well as the relations among them. The framework is lifted in that it does not require to first ground the relational MDP and then exhaustively check all possible paths, but rather works at a more abstract relational level where variables are only instantiated whenever needed. The resulting algorithm is based on the relational Bellman operator REBEL. It is quite complex but manages to rather compactly compress enormous spaces of ground states in a couple of 10s of rules. The algorithm has not yet been optimized, and one route for further work is to combine pCTL-REBEL with expressive first order decision diagrams (Wang et al. 2008) to gain efficiency and to further compress the abstract states. Another route for further work is to explore the use of pCTL-REBEL for reasoning and reinforcement learning in safety-critical contexts.

Acknowledgements This work was supported by the FNRS-FWO joint programme under EOS No. 30992574. It has also received funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme, the EU H2020 ICT48 project "TAILOR" under contract #952215, the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and the KU Leuven Research fund.

References

Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., & Topcu, U. (2018). Safe reinforcement learning via shielding. In: *Proceedings of the 32nd 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, (pp. 2669–2678).

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., & Mané, D. (2016). Concrete problems in AI safety. arXiv:1606.06565.

Bagheri Hariri, B., Calvanese, D., De Giacomo, G., Deutsch, A., & Montali, M. (2013). *Verification of relational data-centric dynamic systems with external services* (Vol. '13, pp. 163–174). PODS. https://doi.org/10.1145/2463664.2465221

Baier, C., & Katoen, J. P. (2008). *Principles of model checking (representation and mind series)*. The MIT Press.

Belardinelli, F., Lomuscio, A., & Patrizi, F. (2011). Verification of deployed artifact systems via data abstraction. In G. Kappel, Z. Maamar, & H. R. Motahari-Nezhad (Eds.), *Service-oriented computing* (pp. 142–156). Springer.

Belardinelli, F., Lomuscio, A., & Patrizi, F. (2012). An abstraction technique for the verification of artifact-centric systems. In *Proceedings of the thirteenth international conference on principles of knowledge representation and reasoning, KR* (pp. 319–328). AAAI Press.

Belardinelli, F., Lomuscio, A., & Patrizi, F. (2013). Verification of agent-based artifact systems. CoRR. arXiv:1301.2678

Boutilier, C., Reiter, R., & Price, B. (2001). Symbolic dynamic programming for first-order mdps. In: *Proceedings of the 17th international joint conference on artificial intelligence* (vol. 1, pp. 690–697). Morgan Kaufmann Publishers Inc. IJCAI'01. http://dl.acm.org/citation.cfm?id=1642090.1642184

Calvanese, D., Giacomo, G. D., Montali, M., & Patrizi, F. (2018). First-order μ-calculus over generic transition systems and applications to the situation calculus. *Information and Computation*, *259*, 328 – 347. https://doi.org/10.1016/j.ic.2017.08.007. 22nd International Symposium on Temporal Representation and Reasoning.

de Alfaro, L., & Roy, P. (2007). Magnifying-lens abstraction for Markov decision processes. In W. Damm & H. Hermanns (Eds.), *Computer Aided Verification* (pp. 325–338). Springer.
De Giacomo, G., Lespérance, Y., & Patrizi, F. (2012). Bounded situation calculus action theories and decidable verification. In *Proc of KR 12*.
De Giacomo, G., Lespérance, Y., & Patrizi, F. (2015). Bounded situation calculus action theories. CoRR. http://arxiv.org/abs/1509.02012
De Giacomo, G., Iocchi, L., Favorito, M., & Patrizi, F. (2019). Foundations for restraining bolts: Reinforcement learning with ltlf/ldlf restraining specifications. *Proceedings of the International Conference on Automated Planning and Scheduling*, *29*(1), 128–136. https://ojs.aaai.org/index.php/ICAPS/article/view/3549
de Salvo Braz, R., Amir, E., & Roth, D. (2005). Lifted first-order probabilistic inference. In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence* (pp. 1319–1325). Edinburgh, Scotland. Morgan Kaufmann Publishers Inc. San Francisco
De Raedt, L., Kersting, K., Natarajan, S., & Poole, D. (2016). Statistical relational artificial intelligence: Logic, probability, and computation. *Synthesis Lectures on Artificial Intelligence and Machine Learning, 10*(2), 1–189. https://doi.org/10.2200/S00692ED1V01Y201601AIM032
Dehnert, C., Junges, S., Katoen, J. P., & Volk, M. (2017). A storm is coming: A modern probabilistic model checker. In R. Majumdar & V. Kunčak (Eds.), *Computer aided verification* (pp. 592–600). Springer.
Driessens, K., & Džeroski, S. (2004). Integrating guidance into relational reinforcement learning. *Machine Learning, 57*, 271–304. https://doi.org/10.1023/B:MACH.0000039779.47329.3a
Džeroski, S., De Raedt, L., & Driessens, K. (2001). Relational Reinforcement Learning. *Machine learning, 43*(1–2), 7–52.
Ferilli, S., Fanizzi, N., Mauro, N. D., & Basile, T. M. A. (2002). Efficient theta-subsumption under object identity. In *In atti del workshop AI*IA su apprendimento automatico*.
Forejt, V., Kwiatkowska, M., Norman, G., & Parker, D. (2011). *Automated verification techniques for probabilistic systems* (pp. 53–113). Springer. https://doi.org/10.1007/978-3-642-21455-4_3
Fulton, N., & Platzer, A. (2018). Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In *AAAI* (pp. 6485–6492). https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17376
Gabbay, D. M. (2003). *Many-dimensional modal logics: Theory and applications*. Elsevier North Holland.
Garcia, J., & Fernández, F. (2015). A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research, 16*, 1437–1480.
Giacomo, G. D. (2019). Queryable self-deliberating dynamic systems. iJCAI. https://www.cse.ust.hk/pg/seminars/S19/giacomo.html
Giunchiglia, F., & Traverso, P. (2000). Planning as model checking. In S. Biundo & M. Fox (Eds.), *Recent advances in AI planning* (pp. 1–20). Springer.
Haddad, S., & Monmege, B. (2014). *Reachability in MDPs: Refining convergence of value iteration* (Vol. 8762, pp. 125–137). Springer. https://doi.org/10.1007/978-3-319-11439-2_10
Hahn, E. M., Li, Y., Schewe, S., Turrini, A., & Zhang, L. (2014). *IscasMC: A web-based probabilistic model checker* (Vol. 8442, pp. 312–317). Springer.
Hasanbeig, M., Kantaros, Y., Abate, A., Kroening, D., Pappas, G. J., & Lee, I. (2019). Reinforcement learning for temporal logic controlsynthesis with probabilistic satisfaction guarantees. In *2019 IEEE 58th conference on decision and control (CDC)* (pp. 5338–5343).
He, K., Lahijanian, M., Kavraki, L. E., & Vardi, M. Y. (2015). Towards manipulation planning with temporal logic specifications. In *2015 IEEE international conference on robotics and automation (ICRA)* (pp. 346–352). https://doi.org/10.1109/ICRA.2015.7139022
Jansen, N., Könighofer, B., Junges, S., Serban, A., & Bloem, R. (2020). Safe reinforcement learning using probabilistic shields. In I. Konnov, L. Kovacs (Eds.), *31st international conference on concurrency theory, CONCUR 2020, Schloss Dagstuhl–Leibniz–Zentrum fur informatik GmbH* (pp. 31–316). Dagstuhl Publishing. https://doi.org/10.4230/LIPIcs.CONCUR.2020.3
Kattenbelt, M., Kwiatkowska, M., Norman, G., & Parker, D. (2008). Game-based probabilistic predicate abstraction in prism. *Electronic Notes in Theoretical Computer Science*, *220*3, 5–21, https://doi.org/10.1016/j.entcs.2008.11.016. Proceedings of the Sixth Workshop on Quantitative Aspects of Programming Languages (QAPL 2008).
Kersting, K. (2012). Lifted probabilistic inference. In *ECAI* (pp. 33–38).
Kersting, K., & De Raedt, L. (2004). Logical Markov decision programs and the convergence of logical td(λ). In R. Camacho, R. King, & A. Srinivasan (Eds.), *Inductive logic programming* (pp. 180–197). Springer.

Kersting, K., Otterlo, M. V., & De Raedt, L. (2004). Bellman goes relational. In *Proceedings of the 21st international conference on machine learning*. ACM, ICML '04 (p. 59). https://doi.org/10.1145/1015330.1015401

Kwiatkowska, M., Norman, G., & Parker, D. (2011). In G. Gopalakrishnan & S. Qadeer (Eds.), *PRISM 4.0: Verification of probabilistic real-time systems* (Vol. 6806, pp. 585–591). Springer.

Lahijanian, M., Andersson, S. B., & Belta, C. (2012). Temporal logic motion planning and control with probabilistic satisfaction guarantees. *IEEE Transactions on Robotics, 28*(2), 396–409. https://doi.org/10.1109/TRO.2011.2172150

Leonetti, M., Iocchi, L., & Patrizi, F. (2012). Automatic generation and learning of finite-state controllers. In A. Ramsay & G. Agre (Eds.), *Artificial intelligence: Methodology, systems, and applications* (pp. 135–144). Springer.

Maly, M. R., Lahijanian, M., Kavraki, L. E., Kress-Gazit, H., & Vardi, M. Y. (2013). Iterative temporal motion planning for hybrid systems in partially unknown environments. In *Proceedings of the 16th international conference on hybrid systems: Computation and control, association for computing machinery* (pp. 353–362). HSCC '13. https://doi.org/10.1145/2461328.2461380

Marthi, B. (2007). Automatic shaping and decomposition of reward functions. In *Proceedings of the 24th international conference on machine learning, association for computing machinery* (pp. 601–608). ICML '07. https://doi.org/10.1145/1273496.1273572

Martínez, D., Alenyç, G., & Torras, C. (2017). Relational reinforcement learning with guided demonstrations. *Artificial Intelligence*, *247*, 295 – 312. https://doi.org/10.1016/j.artint.2015.02.006. Special Issue on AI and Robotics.

Mason, G., Calinescu, R., Kudenko, D., & Banks, A. (2018). *Assurance in reinforcement learning using quantitative verification* (pp. 71–96). Springer. https://doi.org/10.1007/978-3-319-66790-4_5

McMillan, K. L. (1993). *Symbolic model checking* (pp. 25–60). Springer. https://doi.org/10.1007/978-1-4615-3190-6_3

Nienhuys-Cheng, S. H., & Wolf, R. (1997). *Foundations of inductive logic programming*. Springer.

Otterlo, M. V. (2004). Reinforcement learning for relational MDPS. In *Proceedings of the machine learning conference of Belgium and the Netherlands*.

Pecka, M., & Svoboda, T. (2014). Safe exploration techniques for reinforcement learning—An overview. In J. Hodicky (Ed.), *Modelling and simulation for autonomous systems* (pp. 357–375). Springer.

Roy, P., Parker, D., Norman, G., & De Alfaro, L. (2008). Symbolic magnifying lens abstraction in Markov decision processes (pp. 3–112). https://doi.org/10.1109/QEST.2008.41.

Sanner, S., & Boutilier, C. (2009). Practical solution techniques for first-order mdps. *Artificial Intelligence, 173*(5), 748–788. https://doi.org/10.1016/j.artint.2008.11.003. Advances in Automated Plan Generation

Slaney, J., & Thiébaux, S. (2001). Blocks world revisited. *Artificial Intelligence, 125*(1), 119–153. https://doi.org/10.1016/S0004-3702(00)00079-5

Sprauel, J., Kolobov, A., & Teichteil-Königsbuch, F. (2014). Saturated path-constrained mdp: Planning under uncertainty and deterministic model-checking constraints. In *28th AAAI conference on artificial intelligence*. AAAI Press. https://www.microsoft.com/en-us/research/publication/saturated-path-constrained-mdp-planning-uncertainty-deterministic-model-checking-constraints/

Teichteil-Königsbuch, F. (2012). Path-Constrained Markov Decision Processes: bridging the gap between probabilistic model-checking and decision-theoretic planning. In *20th European conference on artificial intelligence (ECAI 2012)*. MONTPELLIER. https://hal-onera.archives-ouvertes.fr/hal-01060349

Van den Broeck, G., Taghipour, N., Meert, W., Davis, J., & De Raedt, L. (2011). Lifted probabilistic inference by first-order knowledge compilation. In *Proceedings of the 22nd international joint conference on artificial intelligence, AAAI Press/international joint conferences on artificial intelligence, Menlo* (pp. 2178–2185).

Wang, C., Joshi, S., & Khardon, R. (2008). First order decision diagrams for relational MDPs. *Journal of Artificial Intelligence Research, 31*, 431–472.

Yoon, S. W., Fern, A., & Givan, R. (2012). Inductive policy selection for first-order mdps. arXiv:1301.0614.

Authors and Affiliations

Wen-Chi Yang[1] · Jean-François Raskin[2] · Luc De Raedt[3]

Jean-François Raskin
jean-francois.raskin@ulb.be

Luc De Raedt
luc.deraedt@kuleuven.be

[1] Department of Computer Science, KU Leuven, Celestijnenlaan 200a, Box 2402, 3001 Leuven, Belgium

[2] Université libre de Bruxelles, Campus de la Plaine, CP212, 1050 Bruxelles, Belgium

[3] Centre for Applied Autonomous Sensor Systems, Örebro University, Örebro, Sweden