Counterexample Explanation by Learning Small Strategies in Markov Decision Processes

Tomáš Brázdil[1], Krishnendu Chatterjee[2], Martin Chmelík[2], Andreas Fellner[2], and Jan Křetínský[2](✉)

[1] Masaryk University, Brno, Czech Republic
[2] IST, Klosterneuburg, Austria
jan.kretinsky@ist.ac.at

Abstract. For deterministic systems, a counterexample to a property can simply be an error trace, whereas counterexamples in probabilistic systems are necessarily more complex. For instance, a set of erroneous traces with a sufficient cumulative probability mass can be used. Since these are too large objects to understand and manipulate, compact representations such as subchains have been considered. In the case of probabilistic systems with non-determinism, the situation is even more complex. While a subchain for a given strategy (or scheduler, resolving non-determinism) is a straightforward choice, we take a different approach. Instead, we focus on the strategy itself, and extract the most important decisions it makes, and present its succinct representation.

The key tools we employ to achieve this are (1) introducing a concept of importance of a state w.r.t. the strategy, and (2) learning using decision trees. There are three main consequent advantages of our approach. Firstly, it exploits the quantitative information on states, stressing the more important decisions. Secondly, it leads to a greater variability and degree of freedom in representing the strategies. Thirdly, the representation uses a self-explanatory data structure. In summary, our approach produces more succinct and more explainable strategies, as opposed to e.g. binary decision diagrams. Finally, our experimental results show that we can extract several rules describing the strategy even for very large systems that do not fit in memory, and based on the rules explain the erroneous behaviour.

1 Introduction

The standard models for dynamic stochastic systems with both probabilistic and nondeterministic behaviour are *Markov decision processes* (MDPs) [1–3]. They are widely used in verification of probabilistic systems [4,5] in several ways. Firstly, in concurrent probabilistic systems, such as communication protocols, the nondeterminism arises from scheduling [6,7]. Secondly, in probabilistic systems operating in open environments, such as various stochastic reactive systems, nondeterminism arises from environmental inputs [8,9]. Thirdly, for underspecified probabilistic systems, a controller is synthesized, resolving the nondeterminism in a way that optimizes some objective, such as energy consumption or time constraints in embedded systems [4,5].

© Springer International Publishing Switzerland 2015
D. Kroening and C.S. Păsăreanu (Eds.): CAV 2015, Part I, LNCS 9206, pp. 158–177, 2015.
DOI: 10.1007/978-3-319-21690-4_10

In analysis of MDPs, the behaviour under all possible strategies (schedulers, controllers, policies) is examined. For example, in the first two cases, the result of the verification process is either a guarantee that a given property holds under all strategies, or a counterexample strategy. In the third case, either a witness strategy guaranteeing a given property is synthesized, or its non-existence is stated. In all settings, it is desirable that the output *strategies should be "small and understandable"* apart from correct. Intuitively, it is a strategy with a representation small enough for the human debugger to read and understand where the bug is (in the verification setting), or for the programmer to implement in the device (in the synthesis setting). In this paper, we focus on the verification setting and illustrate our approach mainly on probabilistic protocols. Nonetheless, our results immediately carry over to the synthesis setting.

Obtaining a small and simple strategy may be impossible if the strategy is required to be optimal, i.e., in our setting reaching the error state with the highest possible probability. Therefore, there is a trade-off between simplicity and optimality of the strategies. However, in order to debug a system, a simple counterexample or a series thereof is more valuable than the most comprehensive, but incomprehensible counterexample. In practice, a simple strategy reaching the error with probability smaller by a factor of ε, e.g. one per cent, is a more valuable source of information than a huge description of an optimal strategy. Similarly, controllers in embedded devices should strive for optimality, but only as long as they are small enough to fit in the device. In summary, we are interested in finding small and simple close-to-optimal (ε-optimal) strategies.

How can one obtain a small and simple strategy? This seems to require some understanding of the particular system and the bug. How can we do something like that automatically? The approaches have so far been limited to BDD representations of the strategy, or generating subchains representing a subset of paths induced by the strategy. While BDDs provide a succinct representation, they are not well readable and understandable. Further, subchains do not focus on the decisions the strategy makes at all. In contrast, a huge effort has been spent on methods to obtain "understanding" from large sets of data, using *machine learning* methods. In this paper, we propose to extend their use in verification, namely of reachability properties in MDPs, in several ways. Our first aim of using these methods is to efficiently exploit the structure that is present in the models, written in e.g. PRISM language with variables and commands. This structure gets lost in the traditional numerical analysis of the MDPs generated from the PRISM language description. The second aim is to distil more information from the generated MDPs, namely the importance of each decision. Both lead to an improved understanding of the strategy's decisions.

Our Approach. We propose three steps to obtain the desired strategies. Each of them has a positive effect on the resulting size.

(1) Obtaining a (Possibly Partially Defined and Liberal) ε-optimal Strategy. The ε-optimal strategies produced by standard methods, such as value iteration of PRISM [10], may be too large to compute and overly specific. Firstly, as argued in [11], typically only a small fraction of the system needs to be explored in order to find an ε-optimal strategy, whereas most states are reached with only a very

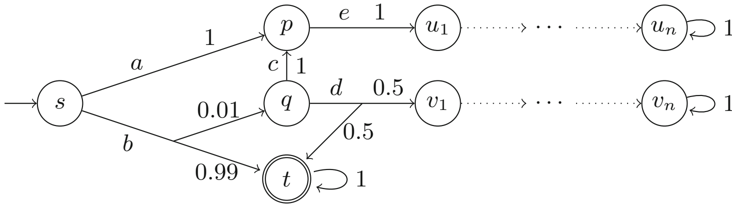


Fig. 1. An MDP M with reachability objective t

small probability. Without much loss, the strategy may not be defined there. For example, in the MDP M depicted in Fig. 1, the decision in q (and v_i's) is almost irrelevant for the overall probability of reaching t from s. Such a partially defined strategy can be obtained using learning methods [11].

Secondly, while the usual strategies prescribe which action to play, *liberal* strategies leave more choices open. There are several advantages of liberal strategies, and similar notions of strategies called permissive strategies have been studied in [12–14]. A liberal strategy, instead of choosing an action in each state, chooses a set of actions to be played uniformly at every state. First, each liberal strategy represents a set of strategies, and thus covers more behaviour. Second, in counter-example guided abstraction-refinement (CEGAR) analysis, since liberal strategies can represent sets of counter-examples, they accelerate the abstraction-refinement loop by ruling out several counter-examples at once. Finally, they also allow for more robust learning of smaller strategies in Step 3. We show that such strategies can be obtained from standard value iteration as well as [11]. Further processing of the strategies in Step 2 and 3 allows liberal strategies as input and preserves liberality in the small representation of the strategy.

(2) Identifying Important Parts of the Strategy. We define a concept of *importance* of a state w.r.t. a strategy, corresponding to the probability of visiting the state by the strategy. Observe that only a fraction of states can be reached while following the strategy, and thus have positive importance. On the unreachable states, with zero importance, the definition of the strategy is useless. For instance, in M, both states p and q must have been explored when constructing the strategy in order to find out whether it is better to take action a or b. However, if the resulting strategy is to use b and d, the information what to do in u_i's is useless. In addition, we consider v_i's to be of zero importance, too, since they are never reached on the way to target.

Furthermore, apart from ignoring states with zero importance, we want to partially ignore decisions that are unlikely to be made (in less important states such as q), and in contrast, stress more the decisions in important states likely to be visited (such as s). Note that this is difficult to achieve in data structures that remember all the stored data exactly, such as BDDs. Of course, we can store decisions in states with importance above a certain threshold. However, we obtain much smaller representations if we allow more variability and reflect the whole quantitative information, as shown in Step 3.

(3) Data Structures for Compact Representation of Strategies. The explicit representation of a strategy by a table of pairs (state, action to play) results in a huge amount of data since the systems often have millions of states. Therefore, a symbolic representation by binary decision diagrams (BDD) looks as a reasonable option. However, there are several drawbacks of using BDDs. Firstly, due to the bit-level representation of the state-action pairs, the resulting BDD is not very readable. Secondly, it is often still too large to be understood by human, for instance due to a bad ordering of the variables. Thirdly, it cannot quantitatively reflect the differences in the importance of states.

Therefore, we propose to use *decision trees* instead, e.g. [15], a structure similar to BDDs, but with nodes labelled by various predicates over the system's variables. They have several advantages. Firstly, they yield an explanation of the decision, as opposed to e.g. neural networks, and thus provide an explanation how the strategy works. Secondly, sophisticated algorithms for their construction, based on entropy, result in smaller representation than BDD, where a good ordering of variables is known to be notoriously difficult to find [4]. Thirdly, as suggested in Step 2, they allow for less probable remembering of less stressed data if this sufficiently simplifies the tree and decreases its size. Finally, the major drawback of decision trees in machine learning—frequent overfitting of the training data—is not an issue in our setting since the tree is not used for classification of test data, but only of the training data.

Summary of Our Contribution. In summary our contributions are as follows:

- We provide a method for obtaining succinct representation of ε-optimal strategies as decision trees. The method is based on a new concept of importance measure and on well-established machine learning techniques.
- Experimental data show that even for some systems larger than the available memory, our method yields trees with only several dozens of nodes.
- We illustrate the understandability of the representation on several examples from PRISM benchmarks [16], reading off respective bug explanations.

Related Work. In artificial intelligence, compact (factored) representations of MDP structure have been developed using dynamic Bayesian networks [17,18], probabilistic STRIPS [19], algebraic decision diagrams [20], and also decision trees [17]. Formalisms used to represent MDPs can, in principle, be used to represent values and policies as well. In particular, variants of decision trees are probably the most used [17,21,22]. For a detailed survey of compact representations see [23]. In the context of verification, MDPs are often represented using variants of (MT)BDDs [24–26], and strategies by BDDs [27].

Decision trees have been used in connection with real-time dynamic programming and reinforcement learning [28,29]. Learning a compact decision tree representation of a policy has been investigated in [30] for the case of body sensor networks, but the paper aims at a completely different application field (a simple model of sensor networks as opposed to generic PRISM models), uses different objectives (discounted rewards), and does not consider the importance of a state that, as we show, may substantially decrease sizes of resulting policies.

Our results are related to the problem of computing minimal/small counterexamples in probabilistic verification. Most papers concentrate on solving this problem for Markov chains and linear-time properties [31–34], branching-time properties [35–37], and in the context of simulation [38]. A couple of tools have been developed for probabilistic counterexample generation, namely DiPro [39] and COMICS [40]. For a detailed survey see [41]. While previous approaches focus on presenting diagnostic paths forming the counterexample, our approach focuses on decisions made by the respective strategy.

Concerning MDPs, [33] uses mixed integer linear programming to compute minimal critical sub-systems, i.e. whole sub-MDPs as opposed to a compact representation of "right" decisions computed by our methods. [42] uses a directed on-the-fly search to compute sets of most probable diagnostic paths (which somehow resembles our notion of importance), but the paths are encoded explicitly by AND/OR trees as opposed to our use of decision trees. Neither of these papers takes advantage of an internal structure of states and their methods substantially differ from ours. The notion of paths encoded as AND/OR trees has also been studied in [43] to represent probabilistic counter-examples visually as fault trees, and then derive causal (the cause and effect) relationship between events. [44] develops abstraction-based framework for model-checking MDPs based on games, which allows to trade compactness for precision, but does not give a procedure for constructing (a compact representation of) counterexample strategies. [45,46] computes a smallest set of guarded commands (of a PRISM-like language) that induce a critical subsystem, but, unlike our methods, does not provide a compact representation of actual decisions needed to reach an erroneous state; moreover, there is not always a command based counterexample.

Counter-examples play a crucial role in CEGAR analysis of MDPs, and have been widely studied, such as, game-based abstraction refinement [47]; non-compositional CEGAR approach for reachability [48] and safe-pCTL [49]; compositional CEGAR approach for safe-pCTL and qualitative logics [38,50]; and abstraction-refinement for quantitative properties [51,52]. All of these works only consider a single strategy represented explicitly, whereas our approach considers a succinct representation of a set of strategies, and can accelerate the abstraction-refinement loop.

2 Preliminaries

We use $\mathbb{N}$, $\mathbb{Q}$, and $\mathbb{R}$ to denote the sets of positive integers, rational and real numbers, respectively. The set of all rational probability distributions over a finite set X is denoted by $\mathit{Dist}(X)$. Further, $d \in \mathit{Dist}(X)$ is Dirac if $d(x) = 1$ for some $x \in X$. Given a function $f : X \to \mathbb{R}$, we write $\arg\max_{x \in X} f(x) = \{x \in X \mid f(x) = \max_{x' \in X} f(x')\}$.

Markov Chains. A *Markov chain* is a tuple $M = (L, P, \mu)$ where L is a finite set of locations, $P : L \to \mathit{Dist}(L)$ is a probabilistic transition function, and $\mu \in \mathit{Dist}(L)$ is the initial probability distribution. We denote the respective unique probability measure for M by $\mathbb{P}$.

Markov Decision Processes. A *Markov decision process* (MDP) is a tuple $G = (S, A, Act, \delta, \hat{s})$ where S is a finite set of states, A is a finite set of actions, $Act : S \to 2^A \setminus \{\emptyset\}$ assigns to each state s the set $Act(s)$ of actions enabled in s, $\delta : S \times A \to Dist(S)$ is a probabilistic transition function that, given a state and an action, gives a probability distribution over the successor states, and $\hat{s}$ is the initial state. A *run* in G is an infinite alternating sequence of states and actions $\omega = s_1 a_1 s_2 a_2 \cdots$ such that for all $i \geq 1$, we have $a_i \in Act(s_i)$ and $\delta(s_i, a_i)(s_{i+1}) > 0$. A *path* of length k in G is a finite prefix $w = s_1 a_1 \cdots a_{k-1} s_k$ of a run in G.

Strategies and Plays. Intuitively, a strategy (or a policy) in an MDP G is a "recipe" to choose actions. Formally, a strategy is a function $\sigma : S \to Dist(A)$ that given the current state of a play gives a probability distribution over the enabled actions.[1] In general, a strategy may randomize, i.e. return non-Dirac distributions. A strategy is *deterministic* if it gives a Dirac distribution for every argument.

A *play* of G determined by a strategy σ and a state $\bar{s} \in S$ is a Markov chain $G^\sigma_{\bar{s}}$ where the set of locations is S, the initial distribution μ is Dirac with $\mu(\bar{s}) = 1$ and

$$P(s)(s') = \sum_{a \in A} \sigma(s)(a) \cdot \delta(s, a)(s') \,.$$

The induced probability measure is denoted by $\mathbb{P}^\sigma_{\bar{s}}$ and "almost surely" or "almost all runs" refers to happening with probability 1 according to this measure. We usually write $\mathbb{P}^\sigma$ instead of $\mathbb{P}^\sigma_{\hat{s}}$ (here $\hat{s}$ is the initial state of G).

Liberal Strategies. A *liberal strategy* is a function $\varsigma : S \to 2^A$ such that for every $s \in S$ we have that $\emptyset \neq \varsigma(s) \subseteq Act(s)$. Given a liberal strategy ς and a state s, an action $a \in Act(s)$ is *good* (in s w.r.t. ς) if $a \in \varsigma(s)$, and *bad* otherwise. Abusing notation, we denote by ς the strategy that to every state s assigns the uniform distribution on $\varsigma(s)$ (which, in particular, allows us to use G^ς_s, $\mathbb{P}^\varsigma_s$ and apply the notion of ε-optimality to ς).

Reachability Objectives. Given a set $F \subseteq S$ of *target states*, we denote by $\Diamond F$ the set of all runs that visit a state of F. For a state $s \in S$, the *maximal reachability probability* (or simply *value*) in s, is $Val(s) := \max_\sigma \mathbb{P}^\sigma_s[\Diamond F]$. Given $\epsilon \geq 0$, we say that a strategy σ is *ε-optimal* if $\mathbb{P}^\sigma[\Diamond F] \geq Val(\hat{s}) - \varepsilon$, and we call a 0-optimal strategy *optimal*.[2] To avoid overly technical notation, we assume that states of F, subject to the reachability objective, are absorbing, i.e. for all $s \in F, a \in Act(s)$ we have $\delta(s, a)(s) = 1$.

End Components. A non-empty set $S' \subseteq S$ is an *end component* (EC) of G if there is $Act' : S' \to 2^A \setminus \{\emptyset\}$ such that (1) for all $s \in S'$ we have $Act'(s) \subseteq Act(s)$,

[1] In general, a strategy may be history dependent. However, for objectives considered in this paper, *memoryless* strategies (depending on the last state visited) are sufficient. Therefore, we only consider memoryless strategies in this paper.

[2] For every MDP, there is a memoryless deterministic optimal strategy, see e.g. [2].

(2) for all $s \in S'$, we have $a \in Act'(s)$ iff $\delta(s,a) \in Dist(S')$, and (3) for all $s, t \in S'$ there is a path $\omega = s_1 a_1 \cdots a_{k-1} s_k$ such that $s_1 = s$, $s_k = t$, and $s_i \in S', a_i \in Act'(s_i)$ for every i. An end component is a *maximal end component* (MEC) if it is maximal with respect to the subset ordering. Given an MDP, the set of MECs is denoted by MEC. Given a MEC, actions of $Act'(s)$ and $Act(s) \setminus Act'(s)$ are called *internal* and *external* (in state s), respectively.

3 Computing ε-Optimal Strategies

There are many algorithms for solving quantitative reachability in MDPs, such as the value iteration, the strategy improvement, linear programming based methods etc., see [2]. The main method implemented in PRISM is the value iteration, which successively (under)approximates the value $Val(s,a) = \sum_{s' \in A} \delta(s,a)(s') \cdot Val(s')$ of every state-action pair (s,a) by a value $V(s,a)$, and stops when the approximation is good enough. Denoting by $V(s) := \max_{a \in Act(s)} V(s,a)$, every step of the value iteration *improves* the approximation $V(s,a)$ by assigning $V(s,a) := \sum_{s' \in S} \delta(s,a)(s') \cdot V(s')$ (we start with V such that $V(s) = 1$ if $s \in F$, and $V(s) = 0$ otherwise).

The disadvantage of the standard value iteration (and also most of the above mentioned traditional methods) is that it works with the whole state space of the MDP (or at least with its reachable part). For instance, consider states u_i, v_i of Fig. 1. The paper [11] adapts methods of bounded real-time dynamic programming (BRTDP, see e.g. [53]) to speed up the computation of the value iteration by improving $V(s,a)$[3] only on "important" state-action pairs identified by simulations.

Even though RTDP methods may substantially reduce the size of an ε-optimal strategy, its explicit representation is usually large and difficult to understand. Thus we develop succinct representations of strategies, based on decision trees, that will reduce the size even further and also provide a human readable representation. Even though the above methods are capable of yielding *deterministic* ε-optimal strategies, that can be immediately fed into machine learning algorithms, we found it advantageous to give the learning algorithm more freedom in the sense that if there are more ε-optimal strategies, we let the algorithm choose (uniformly). This is especially useful within MECs where many actions have the same value. Therefore, we extract *liberal* ε-optimal strategies from the value approximation V, output either by the value iteration or BRTDP.

Computing Liberal ε-Optimal Strategies. Let us show how to obtain a liberal strategy ς from the value iteration, or BRTDP. For simplicity, we start with MDP without MECs.

MDP without End Components. We say that $V : S \times A \to [0,1]$ is a *valid ε-underapproximation* if the following conditions hold:

1. $V(s,a) \leq Val(s,a)$ for all $s \in S$ and $a \in A$
2. $Val(\hat{s}) - V(\hat{s}) \leq \varepsilon$
3. $V(s,a) \leq \sum_{s' \in S} \delta(s,a)(s') \cdot V(s')$ for all $s \in S$ and $a \in Acts$

[3] Here we use V for the lower approximation denoted by V_L in [11].

The outputs V of both the value iteration, and BRTDP are valid ε-underapproximations. We define a liberal strategy ς^V by $\varsigma^V(s) = \arg\max_{a \in Act(s)} V(s,a)$ for all $s \in S$.[4]

Lemma 1. *For every $\varepsilon > 0$ and a valid ε-underapproximation V, ς^V is ε-optimal.*[5]

General MDP. For MDPs with end components we have to extend the definition of the valid ε-underapproximation. Given a MEC $S' \subseteq S$, we say that $(s,a) \in S \times A$ is *maximal-external in* S' if $s \in S'$, $a \in Act(s)$ is external and $V(s,a) \geq V(s',a')$ for all $s' \in S'$ and $a' \in Act(s')$. A state $s' \in S'$ is an *exit* (of S') if (s,a) is maximal-external in S' for some $a \in Act(s)$. We add the following condition to the valid ε-underapproximation:

4. Each MEC $S' \subseteq S$ has at least one exit.

Now the definition of ς^V is also more complicated:

- For every $s \in S$ which is *not* in any MEC, we put $\varsigma^V(s) = \arg\max_{a \in Act(s)} V(s,a)$.
- For every $s \in S$ which *is* in a MEC S',
  - if s is an exit, then $\varsigma^V(s) = \{a \in Act(s) \mid (s,a) \text{ is maximal-external in } S'\}$
  - otherwise, $\varsigma^V(s) = \{a \in Act(s) \mid a \text{ is internal}\}$

Using these extended definitions, Lemma 1 remains valid. Further, note that $\varsigma^V(s)$ is defined even for states with trivial underapproximation $V(s) = 0$, for instance a state s that was never subject to any value iteration improvement. Then the values $\varsigma(s)$ may not be stored explicitly, but follow implicitly from *not* storing any $V(s)$, thus assuming $V(s,\cdot) = 0$.

4 Importance of Decisions

Note that once we have computed an ε-optimal liberal strategy ς, we may, in principle, compute a compact representation of ς (using e.g. BDDs), and obtain a strategy with possibly smaller representation than above.

However, we go one step further as follows. Given a liberal strategy ς and a state $s \in S$, we define the *importance* of s by

$$Imp^{\varsigma}(s) := \mathbb{P}^{\varsigma}[\Diamond s \mid \Diamond F]$$

the probability of visiting s conditioned on reaching F (afterwards). Intuitively, the importance is high for states where a good decision can help to reach the target.[6]

[4] Furthermore, one could consider liberal strategies playing also ε-optimal actions. However, our experiments did not prove better performance.

[5] Intuitively this means that randomizing among good actions of ε-optimal strategies preserves ε-optimality in the reachability setting (in contrast to other settings, e.g. with parity objectives).

[6] Instead of the conditional probability of reaching s, we could consider the conditional expected number of visits of s. We discuss the differences and compare the efficiency together with the case of no conditioning on reaching the target in Sect. 6.

Example 1. For the MDP of Fig. 1 with the objective $\Diamond\{t\}$ and a strategy ς choosing b, we have $Imp^{\varsigma}(s) = 1$ and $Imp^{\varsigma}(q) = 5/995$. Trivially, $Imp^{\varsigma}(t) = 1$. For all other states, the importance is zero.

Obviously, decisions made in states of zero importance do not affect $\mathbb{P}^{\varsigma}[\Diamond F]$ since these states never occur on paths from $\hat{s}$ to F. However, note that many states of S may be reachable in G^{ς} with positive but negligibly small probability. Clearly, the value of $\mathbb{P}^{\varsigma}[\Diamond F]$ depends only marginally on choices made in these states. Formally, let ς_{Δ} be a strategy obtained from ς by changing each $\varsigma(s)$ with $Imp^{\varsigma}(s) \leq \Delta$ to an arbitrary subset of $Act(s)$. We obtain the following obvious property:

Lemma 2. *For every liberal strategy ς, we have $\lim_{\Delta \to 0} \mathbb{P}^{\varsigma_{\Delta}}[\Diamond F] = \mathbb{P}^{\varsigma}[\Diamond F]$.*

In fact, every $\Delta < \min(\{Imp^{\varsigma}(s) \mid s \in S\} \setminus \{0\})$ satisfies $\mathbb{P}^{\varsigma_{\Delta}}[\Diamond F] = \mathbb{P}^{\varsigma}[\Diamond F]$. But often even larger Δ may give $\mathbb{P}^{\varsigma_{\Delta}}[\Diamond F]$ sufficiently close to $\mathbb{P}^{\varsigma}[\Diamond F]$. Such Δ may be found using e.g. trial and error approach.[7]

Most importantly, we can use the importance of a state to affect the probability that decisions in this state are indeed remembered in the data structure. Data structures with such a feature are used in various learning algorithms. In the next section, we discuss decision trees. Due to this extra variability, which decisions to learn, the resulting decision trees are smaller than BDDs for strictly defined ς_{Δ}.

5 Efficient Representations

Let $G = (S, A, Act, \delta, \hat{s})$ be an MDP. In order to symbolically represent strategies in G, we need to assume that states and actions have some internal structure. Inspired by PRISM language [5], we consider a set $\mathcal{V} = \{v_1, \ldots, v_n\}$ of *integer variables*, each v_i gets its values from a finite domain $Dom(v_i)$. We suppose that $S = \prod_{i=1}^{n} Dom(v_i) \subseteq \mathbb{Z}^n$, i.e. each state is a vector of integers. Further, we assume that the MDP arises as a product of m modules, each of which can separately perform non-synchronizing actions as well as synchronously with other modules perform a synchronizing action. Therefore, we suppose $A \subseteq \bar{A} \times \{0, \ldots, m\}$, where $\bar{A} \subseteq \mathbb{N}$ is a finite set and the second component determines the module performing the action (0 stands for synchronizing actions).[8]

Since a liberal strategy is a function of the form $\varsigma : S \to 2^A$, assigning to each state its good actions, it can be *explicitly* represented as a list of state-action pairs, i.e., as a subset of

$$S \times A = \prod_{i=1}^{n} Dom(v_i) \times \bar{A} \times \{0, 1, \ldots, m\} \tag{1}$$

[7] One may give a theoretical bound on convergence of $\mathbb{P}^{\varsigma_{\Delta}}[\Diamond F]$ to $\mathbb{P}^{\varsigma}[\Diamond F]$ as $\Delta \to 0$, using e.g. Lemma 5.1 of [54]. However, for large MDPs the bound would be impractical.

[8] On the one hand, PRISM does not allow different modules to have local variables with the same name, hence we do not distinguish which module does a variable belong to. On the other hand, while PRISM declares no names for non-synchronizing actions, we want to exploit the connection between the corresponding actions of different copies of the same module.

In addition, standard optimization algorithms implemented in PRISM use an explicit "don't-care" value -2 for action in each unreachable state, meaning the strategy is not defined. However, one could simply not list these pairs at all. Thus a smaller list is obtained, with only the states where ς is defined. Recall that one may also omit states s satisfying $Imp^{\varsigma}(s) = 0$, thus ignoring reachable states with zero probability to reach the target. Further optimization may be achieved by omitting states s satisfying $Imp^{\varsigma}(s) < \Delta$ for a suitable $\Delta > 0$.

5.1 BDD Representation

The explicit set representation can be encoded as a binary decision diagram (BDD). This has been used in e.g. [27,55]. The principle of the BDD representation of a set is that (1) each element is encoded as a string of bits and (2) an automaton, in the form of a binary directed acyclic graph, is created so that (3) the accepted language is exactly the set of the given bit strings. Although BDDs are quite efficient, see Sect. 6, each of these three steps can be significantly improved:

1. Instead of a string of bits describing all variables, a string of integers (one per variable) can be used. Branching is then done not on the value of each bit, but according to an inequality comparing the variable to a constant. This significantly improves the readability.
2. Instead of building the automaton according to a chosen order of bits, we let a heuristic choose the order of the inequalities and the actual constants in the inequalities.
3. Instead of representing the language precisely, we allow the heuristic to choose which data to represent and which not. The likelihood that each datum is represented corresponds to its importance, which we provide as another input.

The latter two steps lead to significantly smaller graphs than BDDs. All this can be done in an efficient way using decision trees learning.

5.2 Representation Using Decision Trees

Decision Trees. A *decision tree* for a domain $\prod_{i=1}^{d} X_i \subseteq \mathbb{Z}^d$ is a tuple $\mathcal{T} = (T, \rho, \theta)$ where T is a finite rooted binary (ordered) tree with a set of inner nodes N and a set of leaves L, ρ assigns to every inner node a predicate of the form $[x_i \sim \mathit{const}]$ where $i \in \{1, \ldots, d\}$, $x_i \in X_i$, $\mathit{const} \in \mathbb{Z}$, $\sim \in \{\leq, <, \geq, >, =\}$, and θ assigns to every leaf a value *good*, or *bad*.[9]

Similarly to BDDs, the language $\mathcal{L}(\mathcal{T}) \subseteq \mathbb{N}^n$ of the tree is defined as follows. For a vector $\bar{\boldsymbol{x}} = (\bar{x}_1, \ldots, \bar{x}_n) \in \mathbb{N}^n$, we find a path p from the root to a leaf such that for each inner node n on the path, the predicate $\rho(n)$ is satisfied by substitution $x_i = \bar{x}_i$ iff the first child of n is on p. Denote the leaf on this particular path by ℓ. Then $\bar{\boldsymbol{x}}$ is in the language $\mathcal{L}(\mathcal{T})$ of $\mathcal{T}$ iff $\theta(\ell) = \mathit{good}$.

[9] There exist many variants of decision trees in the literature allowing arbitrary branching, arbitrary values in the leaves, etc., e.g. [15]. For simplicity, we define only a special suitable subclass.

Example 2. Consider dimension $d = 1$, domain $X_1 = \{1, \ldots, 7\}$. A tree representing a set $\{1, 2, 3, 7\}$ is depicted in Fig. 2. To depict the ordered tree clearly, we use unbroken lines for the first child, corresponding to the satisfied predicate, and dashed line for the second one, corresponding to the unsatisfied predicate.

In our setting, we use the domain $S \times A$ defined by Equation (1) which is of the form $\prod_{i=1}^{n+2} X_i$ where for each $1 \leq i \leq n$ we have $X_i = \mathit{Dom}(v_i)$, $X_{n+1} = \bar{A}$ and $X_{n+2} = \{0, 1, \ldots, m\}$. Here the coordinates $\mathit{Dom}(v_i)$ are considered "unbounded" and, consequently, the respective predicates use inequalities. In contrast, we know the possible values of $\bar{A} \times \{0, 1, \ldots, m\}$ in advance and they are not too many. Therefore, these coordinates are considered "discrete" and the respective predicates use equality. Examples of such trees are given in Sect. 6 in Figs. 4 and 5. Now a decision tree $\mathcal{T}$ for this domain determines a liberal strategy $\varsigma : S \to 2^A$ by $a \in \varsigma(s)$ iff $(s, a) \in \mathcal{L}(\mathcal{T})$.

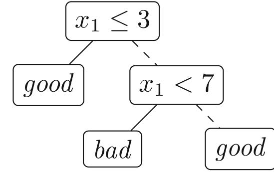


Fig. 2. A decision tree for $\{1, 2, 3, 7\} \subseteq \{1, \ldots, 7\}$

Learning. We describe the process of *learning a training set*, which can also be understood as storing the input data. Given a training sequence (repetitions allowed!) $\boldsymbol{x}^1, \ldots, \boldsymbol{x}^k$, with each $\boldsymbol{x}^i = (x_1^i, \ldots, x_n^i) \in \mathbb{N}^d$, partitioned into the *positive* and *negative* subsequence, the process of learning according to the algorithm ID3 [15,56] proceeds as follows:

1. Start with a single node (root), and assign to it the whole training sequence.
2. Given a node n with a sequence τ,
   - if all training examples in τ are positive, set $\theta(n) = \mathit{good}$ and stop;
   - if all training examples in τ are negative, set $\theta(n) = \mathit{bad}$ and stop;
   - otherwise,
     - choose a predicate with the "highest gain" (with lowest entropy, see e.g. [15, Sects. 3.4.1, 3.7.2]),
     - split τ into sequences satisfying and not satisfying the predicate, assign them to the first and the second child, respectively,
     - go to step 2 for each child.

Intuitively, the predicate with the highest gain splits the sequence so that it maximizes the portion of positive data in the satisfying subsequence and the portion of negative data in the non-satisfying subsequence.

In addition, the final tree can be *pruned*. This means that some leaves are merged, resulting in a smaller tree at the cost of some imprecision of storing (the language of the tree changes). The pruning phase is quite sophisticated, hence for the sake of simplicity and brevity, we omit the details here. We use the standard C4.5 algorithm and refer to [15,57]. In Sect. 6, we comment on effects of parameters used in pruning.

Learning a Strategy. Assume that we already have a liberal strategy $\varsigma : S \to 2^A$. We show how we learn good and bad state-action pairs so that the language of the resulting tree is close to the set of good pairs. The training sequence will be composed of state-action pairs where good pairs are positive examples, and bad pairs are negative ones. Since our aim is to ensure that important states are learnt and not pruned away, we repeat pairs with more important states in the training sequence more frequently.

Formally, for every $s \in S$ and $a \in Act(s)$, we put the pair (s, a) to the training sequence $repeat(s)$-times, where

$$repeat(s) = c \cdot Imp^{\varsigma}(s)$$

for some constant $c \in \mathbb{N}$ (note that $Imp^{\varsigma}(s) \leq 1$). Since we want to avoid exact computation of $Imp^{\varsigma}(s)$, we estimate it using simulations. In practice, we thus run c simulation runs that reach the target and set $repeat(s)$ to be the number of runs where s was also reached.

6 Experiments

In this section, we present the experimental evaluation of the presented methods, which we have implemented within the probabilistic model checker PRISM [5]. All the results presented in this section were obtained on a single Intel(R) Xeon(R) CPU (3.50 GHz) with memory limited to 10 GB.

First, we discuss several alternative options to construct the training data and to learn them in a decision tree. Further, we compare decision trees to other data structures, namely sets and BDDs, with respect to the sizes necessary for storing a strategy. Finally, we illustrate how the decision trees can be used to gain insight into our benchmarks.

6.1 Decision Tree Learning

Generating Training Data. The strategies we work with come from two different sources. Firstly, we consider strategies constructed by PRISM, which can be generated using the explicit or sparse model checking engine. Secondly, we consider strategies constructed by the BRTDP algorithm [11], which are defined on a part of the state space only.

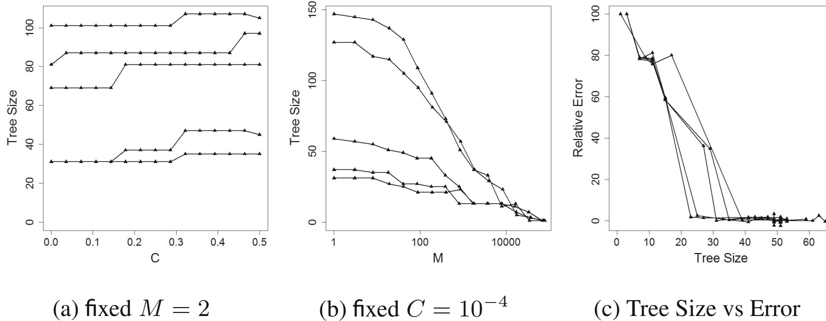
Recall that given a strategy, the training data for the decision trees is constructed from c simulation runs according to the strategy. In our experiments, we found that $c = 10000$ produces good results in all the examples we consider. Note that we stop each simulation as soon as the target or a state with no path to the target state is reached.

Decision Tree Learning in Weka. The decision trees are constructed using the Weka machine learning package [58]. The Weka suite offers various decision tree classifiers. We use the J48 classifier, which is an implementation of the C4.5 algorithm [57]. The J48 classifier offers two parameters to control the pruning that affect the size of the decision tree:

- The leaf size parameter $M \in \mathbb{N}$ determines that each leaf node with less than M instances in the training data is merged with its siblings. Therefore, only values smaller than the number of instances per classification class are reasonable, since higher numbers always result in the trivial tree of size 1.
- The confidence factor $C \in (0, 0.5)$ is used internally for determining the amount of pruning during decision tree construction. Smaller values incur more pruning and therefore smaller trees.

Detailed information and an empirical study of the parameters for J48 is available in [59].

Effects of the Parameters. We illustrate the effects of the parameters C and M on the resulting size of the decision tree on the `mer` benchmark. However, similar behaviour appears in all the examples. Figure 3a and b show the resulting size of the decision tree for several (random) executions. Each line in the plots corresponds to one decision tree, learned with 15 different values of the parameter. The C parameter scales linearly between 0.0001 and 0.5. The M parameter scales logarithmically between 1 and the minimum number of instances per class in the respective training set. The plots in Fig. 3 show that M is an effective parameter in calibrating the resulting tree size, whereas C plays less of a role. Hence, we use $C = 10^{-4}$. Furthermore, since the tree size is monotone in M, the parameter M can be used to retrieve a desired level of detail.



(a) fixed $M = 2$ (b) fixed $C = 10^{-4}$ (c) Tree Size vs Error

Fig. 3. Decision tree parameters

Figure 3c depicts the relation of the tree size to the relative error of the induced strategy. It shows that there is a threshold size under which the tree is not able to capture the strategy correctly anymore and the error rises quickly. Above the threshold size, the error is around 1 %, considered reasonable in order to extract reliable information. This threshold behaviour is observed in all our examples. Therefore, it is sensible to perform a binary search for the highest M ensuring the error at most 1 %.

Table 1. Comparison of representation sizes for strategies obtained from PRISM and from BRTDP computation. Sizes are presented as the number of states for explicit lists of values, the number of nodes for BDDs, and the number of nodes for decision trees (DT). For DT, we display the tree size obtained from the binary search described above. DT Error reports the relative error of the strategy determined by the decision tree (on the induced Markov chain) compared to the optimal value, obtained by model checking with PRISM.

			PRISM				BRTDP			
Example	$\|S\|$	Value	Explicit	BDD	DT	DT Error	Explicit	BDD	DT	DT Error
Firewire	481,136	1.000	479,834	4,233	1	0.000 %	766	4,763[1]	1	0.000 %
Investor	35,893	0.958	28,151	783	27	0.886 %	21,931	2,780	35	0.836 %
Mer_17M	1,773,664	0.200	Memory out				1,887	619	17	0.000 %
Mer_big[2]	Approx. 10^{13}	0.200	Memory out				1,894	646	19	0.692 %
Zeroconf	89,586	0.009	60,463	409	7	0.106 %	1,630	905	7	0.235 %

[1] Note that BDDs represent states in binary form. Therefore, one entry in the explicit state list corresponds to several nodes in the BDD.
[2] We did not measure the state size as the MDP does not fit in memory, but extrapolated it from the linear dependence of model size and one of its parameters, which we could increase to $2^{31} - 1$. The value is obtained from the BRTDP computation.

6.2 Results

First, we briefly introduce the four examples from the PRISM benchmark suite [16], which we tested our method on. Note that the majority of the states in the used benchamrks are non-deterministic, so the strategies are non-trivial in most states.

firewire models the Tree Identify Protocol of the IEEE 1394 High Performance Serial Bus, which is used to transport video and audio signals within a network of multimedia devices. The reachability objective is that one node gets the root and the other one the child role.

investor models a market investor and shares, which change their value probabilistically over time. The reachability objective is that the investor finishes a trade at a time, when his shares are more valuable than some threshold value.

Table 2. Effects of various learning variants on the tree size. Smallest trees computed from PRISM or BRTDP and the average time to compute one number are presented.

Example	$I\Diamond\mathbb{P}$	$I\forall\mathbb{P}$	$I\Diamond\mathbb{E}$	$I\forall\mathbb{E}$	$O\Diamond$	$O\forall$	Avg Time
firewire	1	1	1	1	1	1	45s
investor	27	25	31	35	37	37	135s
mer_17M	17	33	17	29	19	none	314s
mer_big	19	23	23	37	17	none	129s
zeroconf	7	7	7	7	7	17	141s

mer is a mutual exclusion protocol, that regulates the access of two users to two different resources. The protocol should prohibit that both resources are accessed simultaneously.

zeroconf is a network protocol which allows users to choose their IP addresses autonomously. The protocol should detected and prohibit IP address conflict.

For every example, Table 1 shows the size of the state space, the value of the optimal strategy, and the sizes of strategies obtained from explicit model checking by PRISM and by BRTDP, for each discussed data structure.

Learning Variants. In order to justify our choice of the importance function *Imp*, we compare it to several alternatives.

1. When constructing the training data, we can use the importance measure *Imp*, and add states as often as is indicated by its importance (I), or neglect it and simply add every visited state exactly once (O).
2. Further, states on the simulation are learned conditioned on the fact that the target state is reached ($\Diamond$). Another option is to consider all simulations ($\forall$).
3. Finally, instead of the probability to visit the state ($\mathbb{P}$), one can consider the expected number of visits ($\mathbb{E}$).

In Table 2, we report the sizes of the decision trees obtained for the all learning variants. We conclude that our choice (I$\Diamond\mathbb{P}$) is the most useful one.

6.3 Understanding Decision Trees

We show how the constructed decision trees can help us to gain insight into the essential features of the systems.

`zeroconf` example. In Fig. 4 we present a decision tree that is a strategy for `zeroconf` and shows how an unresolved IP address conflict can occur in the protocol. First we present how to read the strategy represented in Fig. 4. Next we show how the strategy can explain the conflict in the protocol. Assume that we are classifying a state-action pair (s, a), where action a is enabled in state s.

1. No matter what the current state s is, the action `rec` is always classified as *bad* according to the root of the tree. Therefore, the action `rec` should be played with positive probability only if all other available actions in the current state are also classified as *bad*.
2. If action a is different from `rec`, the right son of the root node is reached. If action a is different from action `l>0&b=1&ip_mess=1 -> b'=0&z'=0&n1'=min (n1+1,8)&ip_mess'=0` (the whole PRISM command is a single action), then a is classified as *good* in state s. Otherwise, the left son is reached.
3. In node $\mathtt{z} \leq \mathtt{0}$ the classification of action a (that is the action that labels the parent node) depends on the variable valuation of the current state. If the value of var. z is greater than 0, then a is classified as *good* in state s, otherwise it is classified as *bad*.

Action `rec` stands for a network host receiving a reply to a broadcast message, resulting in resolution of an IP address conflict if one is present, which clearly does not help in constructing an unresolved conflict. The action labelling the right son of the root represents the detection of an IP address conflict by an arbitrary network host. This action is only good, if variable z, which is a clock variable, in the current state is greater than 0. The combined meaning of the two nodes is that an unresolved IP address conflict can occur if the conflict is detected too late.

`firewire` example. For `firewire`, we obtain a trivial tree with a single node, labelled *good*. Therefore, playing all available actions in each state guarantees reaching the target almost surely. In contrast to other representations, we have

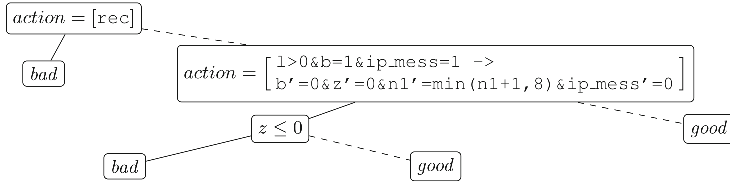


Fig. 4. A decision tree for zeroconf

automatically obtained the information that the network always reaches the target configuration, regardless of the individual behaviour of its components and their interleaving.

`mer` example. In the case of `mer`, there exists a strategy that violates the required property that the two resources are not accessed simultaneously. The decision tree for the `mer` strategy is depicted in Fig. 5. In order to understand how a state is reached, where both resources are accessed at the same time, it is necessary to determine which user accesses which resource in that state.

1. The two tree nodes labelled by 1 explain what resource *user 1* should access. The root node labelled by action `s1=0&r1=0 -> r1'=2` specifies that the request to access *resource* 2 (variable `r1` is set to 2) is classified as *bad*. The only remaining action for *user 1* is to request access to *resource* 1. This action is classified as *good* by the right son of the root node.
2. Analogously, the tree nodes labelled by 2 specify that *user 2* actions should request access to *resource 2* (follows from `s2=0&r2=0 -> r2'=2`). Once *resource 2* is requested it should change its internal state `s2` to 1 (follows from `s2=0&r2=2 -> s2'=1`). It follows, that in the state violating the property, *user 1* has access to *resource 1* and *user 2* to *resource 2*.

The model is supposed to correctly handle such overlapping requests, but fails to do so in a specific case. In order to further debug the model, one has to find the action of the scheduler that causes this undesired behaviour. The lower part of the tree specifies that `u1_request_comm` is a candidate for such an action. Inspecting a snippet of the code of `u1_request_comm` from the PRISM source code (shown below) reveals that in the given situation, the scheduler reacts inappropriately with some probability p.

```
[u1_request_comm]  s=0 & commUser=0 & driveUser!=0 & k<n ->
          (1-p):(s'=1) & (r'=driveUser) & (k'=k+1) +
             p:(s'=-1) & (gc'=true) & (k'=k+1)
```

The remaining nodes of the tree that were not discussed are necessary to reset the situation if the non-faulty part (with probability $1-p$) of the `u1_request_comm` command was executed. It should be noted that executing the faulty `u1_request_comm` action does not lead to the undesired state right away. The action only grants *user 1* access rights in a situation, where he should not get these rights. Only a successive action leads to *user 1* accessing the resource and the undesired state

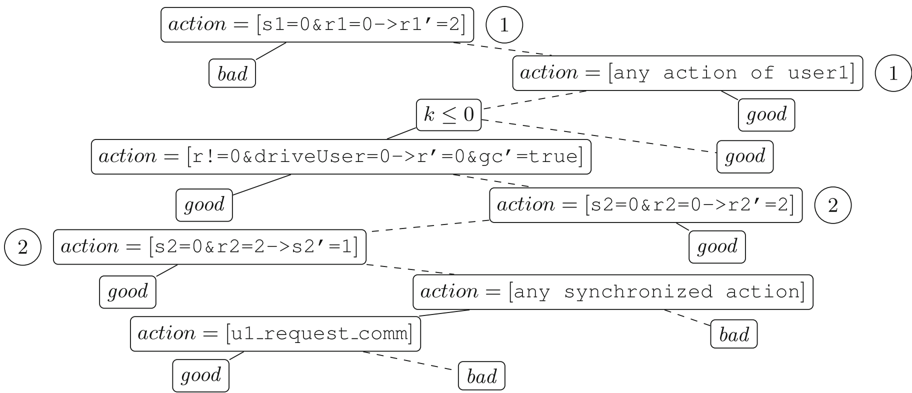


Fig. 5. A decision tree for `mer`

being reached. This is a common type of bug, where the command that triggered an error is not the cause of it.

7 Conclusion

In this work we presented a new approach to represent strategies in MDPs in a succinct and comprehensible way. We exploited machine learning methods to achieve our goals. Interesting directions of future works are to investigate whether other machine learning methods can be integrated with our approach, and to extend our approach from reachability objectives to other objectives (such as long-run average and discounted-sum).

Acknowledgements. This research was funded in part by Austrian Science Fund (FWF) Grant No P 23499-N23, FWF NFN Grant No S11407-N23 (RiSE) and Z211-N23 (Wittgenstein Award), European Research Council (ERC) Grant No 279307 (Graph Games), ERC Grant No 267989 (QUAREM), the Czech Science Foundation Grant No P202/12/G061, and People Programme (Marie Curie Actions) of the European Union's Seventh Framework Programme (FP7/2007–2013) REA Grant No 291734.

References

1. Howard, R.A.: Dynamic Programming and Markov Processes. The MIT press, New York, London, Cambridge (1960)
2. Puterman, M.L.: Markov Decision Processes. Wiley, New York (1994)
3. Filar, J., Vrieze, K.: Competitive Markov Decision Processes. Springer, New York (1997)
4. Baier, C., Katoen, J.-P.: Principles of Model Checking (Representation and Mind Series). The MIT Press, Cambridge (2008)
5. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)

6. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM **42**(4), 857–907 (1995)
7. Vardi, M.: Automatic verification of probabilistic concurrent finite state programs. In: FOCS, pp. 327–338 (1985)
8. Segala, R.: Modeling and verification of randomized distributed real-time systems. Ph.D thesis, MIT Press (1995). Technical report MIT/LCS/TR-676
9. De Alfaro, L.: Formal verification of probabilistic systems. Ph.D thesis, Stanford University (1997)
10. Kwiatkowska, M., Parker, D.: Automated verification and strategy synthesis for probabilistic systems. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 5–22. Springer, Heidelberg (2013)
11. Brázdil, T., Chatterjee, K., Chmelík, M., Forejt, V., Křetínský, J., Kwiatkowska, M., Parker, D., Ujma, M.: Verification of markov decision processes using learning algorithms. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 98–114. Springer, Heidelberg (2014)
12. Bernet, J., Janin, D., Walukiewicz, I.: Permissive strategies: from parity games to safety games. ITA **36**(3), 261–275 (2002)
13. Bouyer, P., Markey, N., Olschewski, J., Ummels, M.: Measuring permissiveness in parity games: mean-payoff parity games revisited. In: Bultan and Hsiung [60] pp. 135–149
14. Dräger, K., Forejt, V., Kwiatkowska, M., Parker, D., Ujma, M.: Permissive controller synthesis for probabilistic systems. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 531–546. Springer, Heidelberg (2014)
15. Mitchell, T.M.: Machine Learning, 1st edn. McGraw-Hill Inc., New York (1997)
16. Kwiatkowska, M., Norman, G., Parker, D.: The PRISM benchmark suite. In: QEST, pp. 203–204 (2012)
17. Boutilier, C., Dearden, R., Goldszmidt, M.: Exploiting structure in policy construction. In: IJCAI-95, pp. 1104–1111 (1995)
18. Kearns, M., Koller, D.: Efficient reinforcement learning in factored MDPs. In: IJCAI, pp. 740–747. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)
19. Kushmerick, N., Hanks, S., Weld, D.: An algorithm for probabilistic least-commitment planning. In: Proceedings of AAAI-94, pp. 1073–1078 (1994)
20. Hoey, J., St-aubin, R., Hu, A., Boutilier, C.: Spudd: stochastic planning using decision diagrams. In: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, pp. 279–288. Morgan Kaufmann (1999)
21. Chapman, D., Kaelbling, L.P.: Input generalization in delayed reinforcement learning: an algorithm and performance comparisons. pp. 726–731. Morgan Kaufmann (1991)
22. Koller, D., Parr, R.: Computing factored value functions for policies in structured MDPs. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, pp. 1332–1339. Morgan Kaufmann (1999)
23. Boutilier, C., Dean, T., Hanks, S.: Decision-theoretic planning: structural assumptions and computational leverage. JAIR **11**, 1–94 (1999)
24. De Alfaro, L., Kwiatkowska, M., Norman, G., Parker, D., Segala, R.: Symbolic model checking of probabilistic processes using MTBDDS and the kronecker representation. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, pp. 395–410. Springer, Heidelberg (2000)
25. Hermanns, H., Kwiatkowska, M., Norman, G., Parker, D., Siegle, M.: On the use of MTBDDs for performability analysis and verification of stochastic systems. J. Log. Algebraic Program. Spec. Issue Probab. Tech. Des. Anal. Syst. **56**(1–2), 23–67 (2003)

26. Miner, A.S., Parker, D.: Symbolic representations and analysis of large probabilistic systems. In: Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.) Validation of Stochastic Systems. LNCS, vol. 2925, pp. 296–338. Springer, Heidelberg (2004)
27. Wimmer, R., Braitling, B., Becker, B., Hahn, E.M., Crouzen, P., Hermanns, H., Dhama, A., Theel, O.: Symblicit calculation of long-run averages for concurrent probabilistic systems. In: QEST, pp. 27–36, IEEE Computer Society, Washington, DC, USA (2010)
28. Boutilier, C., Dearden, R.: Approximating value trees in structured dynamic programming. In: Proceedings of the Thirteenth International Conference on Machine Learning, pp. 54–62 (1996)
29. Pyeatt, L.D.: Reinforcement learning with decision trees. In: The 21st IASTED International Multi-Conference on Applied Informatics (AI 2003), Innsbruck, Austria, pp. 26–31, 10–13 Feb 2003
30. Raghavendra, C.S., Liu, S., Panangadan, A., Talukder, A.: Compact representation of coordinated sampling policies for body sensor networks. In: Proceedings of Workshop on Advances in Communication and Networks (Smart Homes for Tele-Health), pp. 6–10, IEEE (2010)
31. Han, T., Katoen, J.-P., Damman, B.: Counterexample generation in probabilistic model checking. IEEE Trans. Softw. Eng. **35**(2), 241–257 (2009)
32. Andrés, M.E., D'Argenio, P., Van Rossum, P.: Significant diagnostic counterexamples in probabilistic model checking. In: Chockler, H., Hu, A.J. (eds.) HVC 2008. LNCS, vol. 5394, pp. 129–148. Springer, Heidelberg (2009)
33. Wimmer, R., Jansen, N., Ábrahám, E., Katoen, J.-P., Becker, B.: Minimal counterexamples for linear-time probabilistic verification. TCS **549**, 61–100 (2014)
34. Jansen, N., Ábrahám, E., Katelaan, J., Wimmer, R., Katoen, J.-P., Becker, B.: Hierarchical counterexamples for discrete-time markov chains. In: Bultan and Hsiung [60] pp. 443–452
35. Damman, B., Han, T., Katoen, J.-P.: Regular expressions for PCTL counterexamples. In: QEST, pp. 179–188, IEEE Computer Society (2008)
36. Fecher, H., Huth, M., Piterman, N., Wagner, D.: PCTL model checking of markov chains: truth and falsity as winning strategies in games. Perform. Eval. **67**(9), 858–872 (2010)
37. Aljazzar, H., Leue, S.: Directed explicit state-space search in the generation of counterexamples for stochastic model checking. IEEE Trans. Softw. Eng. **36**(1), 37–60 (2010)
38. Komuravelli, A., Păsăreanu, C.S., Clarke, E.M.: Assume-guarantee abstraction refinement for probabilistic systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 310–326. Springer, Heidelberg (2012)
39. Aljazzar, H., Leitner-Fischer, F., Leue, S., Simeonov, D.: DiPro - a tool for probabilistic counterexample generation. In: Groce, A., Musuvathi, M. (eds.) SPIN Workshops 2011. LNCS, vol. 6823, pp. 183–187. Springer, Heidelberg (2011)
40. Jansen, N., Ábráham, E., Volk, M., Wimmer, R., Katoen, J.-P., Becker, B.: The COMICS tool - computing minimal counterexamples for DTMCs. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 349–353. Springer, Heidelberg (2012)
41. Ábrahám, E., Becker, B., Dehnert, C., Jansen, N., Katoen, J.-P., Wimmer, R.: Counterexample generation for discrete-time markov models: an introductory survey. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) SFM 2014. LNCS, vol. 8483, pp. 65–121. Springer, Heidelberg (2014)
42. Aljazzar, H., Leue, S.: Generation of counterexamples for model checking of markov decision processes. In: QEST, pp. 197–206, IEEE Computer Society (2009)

43. Leitner-Fischer, F., Leue, S.: Probabilistic fault tree synthesis using causality computation. IJCCBS **4**(2), 119–143 (2013)
44. Kattenbelt, M., Huth, M.: Verification and refutation of probabilistic specifications via games. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2009, IIT Kanpur, India, pp. 251–262, 15–17 Dec 2009
45. Wimmer, R., Jansen, N., Vorpahl, A., Ábrahám, E., Katoen, J.-P., Becker, B.: High-level counterexamples for probabilistic automata. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 39–54. Springer, Heidelberg (2013)
46. Dehnert, C., Jansen, N., Wimmer, R., Ábrahám, E., Katoen, J.-P.: Fast debugging of PRISM models. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 146–162. Springer, Heidelberg (2014)
47. Kwiatkowska, M.Z., Norman, G., Parker, D.: Game-based abstraction for Markov decision processes. In: QEST, pp. 157–166 (2006)
48. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 162–175. Springer, Heidelberg (2008)
49. Chadha, R., Viswanathan, M.: A counterexample-guided abstraction-refinement framework for Markov decision processes. ACM Trans. Comput. Log. **12**(1), 1 (2010)
50. Chatterjee, K., Chmelík, M., Daca, P.: CEGAR for qualitative analysis of probabilistic systems. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 473–490. Springer, Heidelberg (2014)
51. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability analysis of probabilistic systems by successive refinements. In: De Luca, L., Gilmore, S. (eds.) PROBMIV 2001, PAPM-PROBMIV 2001, and PAPM 2001. LNCS, vol. 2165, pp. 39–56. Springer, Heidelberg (2001)
52. D'Argenio, P.R.: Reduction and refinement strategies for probabilistic analysis. In: Hermanns, H., Segala, R. (eds.) PROBMIV 2002, PAPM-PROBMIV 2002, and PAPM 2002. LNCS, vol. 2399, pp. 57–76. Springer, Heidelberg (2002)
53. McMahan, H.B., Likhachev, M., Gordon, G.J.: Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In: ICML (2005)
54. Brázdil, T., Kiefer, S., Kučera, A.: Efficient analysis of probabilistic programs with an unbounded counter. J. ACM **61**(6), 41:1–41:35 (2014)
55. Von Essen, C., Jobstmann, B., Parker, D., Varshneya, R.: Semi-symbolic computation of efficient controllers in probabilistic environments. Technical report, Verimag (2012)
56. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)
57. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann, San Francisco (1993)
58. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: an update. ACM SIGKDD Explor. Newsl. **11**(1), 10–18 (2009)
59. Drazin, S., Montag, M.: Decision tree analysis using weka. Machine Learning-Project II, University of Miami, pp. 1–3 (2012)
60. Bultan, T., Hsiung, P.-A. (eds.): Automated Technology for Verification and Analysis, ATVA 2011. 9th International Symposium, Taipei, Taiwan, October 11-14, 2011. Proceedings, vol. 6996, LNCS. Springer, Heidelberg (2011)