Statistical Model Checking for Markov Decision Processes

David Henriques[1,2,3] **João Martins**[1,4]
Paolo Zuliani[1] **André Platzer**[1]
Edmund M. Clarke[1]

May 2012
CMU-CS-12-122

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]Computer Science Department, Carnegie Mellon University.
[2] SQIG - Instituto de Telecomunicações
[3] Department of Mathematics, IST - TU Lisbon
[4] CENTRIA, Departamento de Informática, Universidade Nova de Lisboa

This research was sponsored by the GSRC under contract no. 1041377 (Princeton University), the National Science Foundation under contracts no. CNS0926181, no. CNS0931985 and no. CNS1054246, the Semiconductor Research Corporation under contract no. 2005TJ1366, General Motors under contract no. GMCMUCRLNV301, the Air Force Office of Scientific Research (BAA 2011-01), the Office of Naval Research under award no. N000141010188, the Army Research Office under contract no. W911NF-09-1-0273 and the CMU–Portugal Program under grant no SFRH/BD/51846/2012.

Keywords: Statistical Model Checking, Markov Decision Processes, Reinforcement Learning

Abstract

This technical report is a more detailed version of a published paper [14].

Statistical Model Checking (SMC) is a computationally very efficient verification technique based on selective system sampling. One well identified shortcoming of SMC is that, unlike probabilistic model checking, it cannot be applied to systems featuring nondeterminism, such as Markov Decision Processes (MDP). We address this limitation by developing an algorithm that resolves nondeterminism probabilistically, and then uses multiple rounds of sampling and Reinforcement Learning to provably improve resolutions of nondeterminism with respect to satisfying a Bounded Linear Temporal Logic (BLTL) property. Our algorithm thus reduces an MDP to a fully probabilistic Markov chain on which SMC may be applied to give an approximate solution to the problem of checking the probabilistic BLTL property. We integrate our algorithm in a parallelised modification of the PRISM simulation framework. Extensive validation with both new and PRISM benchmarks demonstrates that the approach scales very well in scenarios where symbolic algorithms fail to do so.

1 Introduction

Model Checking [16] (MC) is a successful set of techniques aimed at providing formal guarantees (usually expressed in some form of temporal logic) for models that can be specified as transition systems. There has been a lot of interest in the MC community for extensions of the classical algorithms to probabilistic settings, which are more expressive but significantly harder to analyse. These extensions study the Probabilistic Model Checking (PMC) problem, where the goal is to find the probability that a property holds in some stochastic model.

When solving the PMC problem, it is often possible to trade-off correctness for scalability. There is extensive work on how the PMC problem can be solved through exact techniques [18, 1, 8], which compute correct probability bounds. Exact techniques do, however, rely on reasoning about the entire state space, which is widely considered to be the limiting factor in their applicability to large problems. The complementary approach is known as Statistical Model Checking (SMC), which is based on selectively sampling traces of the system until enough statistical evidence has been found. Although it trades away the iron clad guarantees of PMC for statistical claims, SMC requires comparatively little memory, thus circumventing the most pressing limitation of classical PMC techniques. In addition, sampling is usually very efficient even for large systems.

Currently, one shortcoming of SMC compared to exact methods is that it does not handle systems with nondeterminism, since it is not clear how to resolve nondeterminism during sampling. Thus, SMC can only be directly applied to fully probabilistic systems, such as Markov chains. In this work, we address this problem.

We develop and study a statistical algorithm to enable the application of SMC in Markov decision processes (MDPs), the *de facto* standard for modelling discrete systems exhibiting both stochastic and nondeterministic behaviour. The main difficulty for the PMC problem in MDPs is that it requires properties to hold in *all* resolutions of nondeterminism, or *schedulers*. Properties, expressed in temporal logic and interpreted over traces, often check for bad behaviour in the modelled system. In this case, one would check that, for *all* schedulers, the probability of bad behaviour occurring is less than some small value.

This goal can be reduced to finding the probability under a *most adversarial*, or *optimal scheduler:* one that maximises the probability of satisfying the property. Unfortunately, an exhaustive study of all schedulers would not be computationally feasible. On the other hand, checking only isolated schedulers would not give any significant insight about the behaviour of the system under the optimal resolution of nondeterminism.

Exact methods typically find these optimal schedulers using a fixed point computation that requires propagating information throughout the entire state space whereas our approach does a *guided search* for the most adversarial schedulers. Because of this, we need to consider only a very small fraction of the potential schedulers. We sample from the model under an arbitrary candidate scheduler to estimate how "good" each transition is, i.e., how much it contributes to the satisfaction of the property. Then we *reinforce* good transitions, provably improving the scheduler, and start sampling again with this new candidate. Once we are confident that we have a sufficiently good scheduler, we can use any method for solving the PMC problem for fully probabilistic systems (like classical SMC) to settle the original query. One important advantage of this approach is that,

like in non-probabilistic model checking, if the algorithm finds that the property is false, it provides a *counterexample scheduler*, which can then be used for debugging purposes.

PRISM [18] is a state-of-the-art probabilistic model checker. We implemented our algorithm in Java, using a parallelised version of PRISM's simulation framework for trace generation. This allows us to seamlessly use PRISM's specifications for MDPs. We take care to ensure that our multi-threaded modification of the framework remains statistically unbiased. We apply our algorithm to both the PRISM benchmark suite as well as to new benchmarks and perform an extensive comparison. The results show that the algorithm is highly scalable and efficient. It also runs successfully on problems that are too large to be tackled by PRISM's exact engine.

2 Related Work

Numerical methods compute high precision solutions to the PMC problem [1, 8, 18, 17, 7, 15], but fail to scale for very large systems. Several authors [31, 19, 23, 24] have studied Statistical Model Checking, which handles the PMC problem statistically in fully probabilistic systems. Several implementations [29, 25] have already shown the applicability of SMC. One serious and well identified shortcoming of SMC is that it cannot be applied to even partially nondeterministic systems. The canonical example is a Markov Decision Process (MDP), where one must guarantee some probabilistic property regardless of the resolution of nondeterminism.

We are aware of two attempts at using statistical techniques to solve the PMC problem in nondeterministic settings. In [21], Lassaigne and Peyronnet. deal with planning and verification of monotone properties in MDPs using an adaptation of Kearn's learning algorithm. In addition, in [4], Bogdoll et al. consider this problem with the very restricted form of nondeterminism induced by the commutativity of concurrently executed transitions in compositional settings (spurious nondeterminism).

To solve the general problem, we draw from the Reinforcement Learning literature [27, 6]. Real-Time Dynamic Programming [2] works in a setting similar to PMC. It also uses simulation for the exploration of near-optimal schedulers, but still needs to store the entire system in memory, suffering from the same limitations as numerical PMC techniques.

The scheduler optimisation stage of our algorithm works in a fashion similar to some Monte Carlo methods [27], despite the fact that one maximises the probability of satisfying some path property external to the model and the other maximises some discounted reward inherent to the model. Monte Carlo methods estimate, through simulation, a "fitness" value for each state and then use these values to greedily update their guess as to which is the best scheduler. An similar idea is at the core of our algorithm.

3 Probabilistic Model Checking for MDPs

In this section we lay the necessary formal foundations to define the probabilistic model checking problem.

3.1 State Labeled Markov Decision Processes

Markov decision processes are a popular choice to model discrete state transition systems that are both probabilistic and nondeterministic. Standard statistical model checking does not handle nondeterminism and thus cannot be directly applied to these models. Schedulers are functions used to resolve the nondeterminism in Markov decision processes. A MDP in which nondeterminism has been resolved becomes a fully probabilistic system known as a Markov chain.

In the setting of PMC, it is customary to assume the existence of a state labelling function $\mathcal{L}$ that associates each state with a set of propositions that are true in that state.

Definition 1 (Markov Decision Process)**.** *A* State Labeled Markov Decision Process *(MDP) is a tuple $\mathcal{M} = \langle S, \overline{s}, A, \tau, \mathcal{L} \rangle$ where S is a (finite) set of states, $\overline{s} \in S$ is an initial state, A is a (finite) set of actions, $\tau : S \times A \times S \to [0, 1]$ is a transition function such that for $s \in S, a \in A$, either $\sum_{s' \in S} \tau(s, a, s') = 1$ (a is enabled) or $\sum_{s' \in S} \tau(s, a, s') = 0$ (a is disabled), for each $s \in S$ there exists at least one action enabled from s and $\mathcal{L} : S \to 2^{AP}$ is a labelling function mapping each state to the set of atomic propositions true in that state.*

For each state s and enabled action a, $\tau(s, a, s')$ gives the probability of taking action a in state s and moving to state s'. At least one action needs to be enabled at each state. The transitions are assumed to take one "time step" so there is no notion of real time. Because of this, MDPs are particularly suited for reasoning about the ordering of events without being explicit about their timing.

A scheduler for a MDP resolves the nondeterminism in each state s by providing a distribution over the set of actions enabled in s.

Definition 2 (Scheduler)**.** *A* memoryless scheduler *for a MDP $\mathcal{M}$ is a function $\sigma : S \times A \to [0, 1]$ s.t. $\sum_{a \in A} \sigma(s, a) = 1$ and $\sigma(s, a) > 0$ only if a is enabled in s.*

A scheduler for which either $\sigma(s, a) = 1$ or $\sigma(s, a) = 0$ for all pairs $(s, a) \in S \times A$ is called *deterministic*. In this work, by scheduler, we mean memoryless scheduler.

Discrete time Markov chains are fully probabilistic models. They can be seen as MDPs where the nondeterminism over the actions has been resolved and thus can be thought of as a system that runs without the need of external input.

Definition 3 (Markov Chain)**.** *A* State Labeled discrete time Markov chain *is a tuple $M = \langle S, \overline{s}, A, P, \mathcal{L} \rangle$ where S is a (finite) set of states, $\overline{s} \in S$ is an initial state, A is a (finite) set of action names. $P : S \times A \times S \to [0, 1]$ is a transition function such that for $s \in S$, $\sum_{a \in A} \sum_{s' \in S} P(s, a, s') = 1$ and $\mathcal{L} : S \to 2^{AP}$ is a labelling function mapping each state to a set of atomic propositions that are true in that state.*

The inclusion of action names in this definition is a necessary technical detail. Given a MDP $\mathcal{M}$, any scheduler induces a Markov chain by eliminating nondeterminism; when dealing with Markov chains induced in this way, we will use action names to discriminate which specific MDP action generated each Markov chain transition. Action names have no meaningful semantics otherwise.

There is a set of paths associated with each Markov chain M. A *path* in M, denoted $\pi \in M$, is an infinite sequence $\pi = \overline{s} \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \dots$ of states s.t. for all $i \in \mathbb{N}$, $P(s_i, a_i, s_{i+1}) > 0$. Given a path π, the n-th state of π, denoted π^n, is s_n; the *k-prefix* of π, denoted $\pi|_k$ is the finite subsequence of π that ends in π_k; and the *k-suffix* of π, denoted $\pi|^k$ is the infinite subsequence of π that starts in π^k.

The transition function P induces a canonical probability space over the paths of M as follows. We define the function Pr_f, over finite prefixes: for prefix $\hat{\pi} = \overline{s} \xrightarrow{a_0} s_1 \dots \xrightarrow{a_{k-1}} a_k$, $Pr_f(\hat{\pi}) \triangleq 1$ if $k = 0$, $Pr_f(\hat{\pi}) \triangleq P(\overline{s}, a_0, s_1)P(s_1, a_1, s_2) \dots P(s_{k-1}, a_{k-1}, s_k)$ otherwise. This function extends to a unique measure Pr over the set of (infinite) paths of M [28].

Definition 4 (Markov chain induced by a scheduler)**.** *Given a MDP $\mathcal{M} = \langle S, \overline{s}, A, \tau, \mathcal{L} \rangle$ and a scheduler for $\mathcal{M}$, σ, the* Markov chain induced by *σ, is the Markov chain $\mathcal{M}^\sigma = \langle S, \overline{s}, A, P, \mathcal{L} \rangle$ where $P(s, a, s') \triangleq \sigma(s, a)\tau(s, a, s')$.*

This resolution of nondeterminism will enable us to apply SMC techniques to MDPs, provided wefind a suitable scheduler.

3.2 Bounded Linear Temporal Logic

Linear Temporal Logic (LTL) [22] is a formalism used to reason about the ordering of events without introducing time explicitly. It is interpreted over sequences of states. Each state represents a point in time in which certain propositional assertions hold. Once an event changes the truth value of these assertions, the system moves to a new state.

Sampling and checking of paths needs to be computationally feasible. Since LTL may require paths of arbitrary size, we instead use *Bounded* LTL, which requires only paths of bounded size [32]. In addition, for each path, we may identify a smallest prefix that is sufficient to satisfy or refute the property, which we will call the *minimal sufficient prefix* of the path. This notion is useful in practice to avoid considering unnecessarily long paths. The syntax and semantics of BLTL are summarised in Table 1.

Informally, $\mathbf{F}^{\leq n}\varphi_1$ means "φ_1 will become true within n transitions"; $\mathbf{G}^{\leq n}\varphi_1$ means "φ_1 will be remain true for the next n transitions" and $\varphi_1 \mathbf{U}^{\leq n}\varphi_2$ means "φ_2 will be true within the next n transitions and φ_1 remains true until then". The classical connectives follow the usual semantics.

3.3 Probabilistic and Statistical Model Checking

Let $\mathcal{M}$ be a MDP, φ be a BLTL property and $0 < \theta < 1$ be a rational number. The problem of PMC for these parameters, denoted $P_{\leq\theta}(\varphi)$, lies in deciding whether $\forall\sigma : Pr(\{\pi : \pi \in \mathcal{M}^\sigma, \pi \models \varphi\}) \leq \theta$, that is, "Is the probability of the set of paths of $\mathcal{M}^\sigma$ that satisfy φ at most θ *for all* schedulers σ?"

The formula φ usually encodes an undesirable property, e.g. reaching an error state or violating a critical condition. If we can find the scheduler that maximises the probability of satisfying φ, then we can compare that probability with θ to answer the PMC query, since all other schedulers will achieve a lower value. It can be easily shown that *deterministic* schedulers are sufficient for achieving this maximum probability.

Syntax: $\varphi := p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{F}^{\leq n}\varphi \mid \mathbf{G}^{\leq n}\varphi \mid \varphi\mathbf{U}^{\leq n}\varphi$	
Semantics: $\pi \models \varphi$ iff...	
if φ is...	**Semantics**
p	$p \in \mathcal{L}(\pi^0)$
$\neg\varphi_1$	$\pi \not\models \varphi_1$
$\varphi_1 \vee \varphi_2$	$\pi \models \varphi_1$ or $\pi \models \varphi_2$
$\mathbf{F}^{\leq n}\varphi_1$	$\exists_{i \leq n} : \pi\vert^i \models \varphi_1$
$\mathbf{G}^{\leq n}\varphi_1$	$\forall_{i \leq n} : \pi\vert^i \models \varphi_1$
$\varphi_1\mathbf{U}^{\leq n}\varphi_2$	$\exists_{i \leq n}\forall_{k \leq i} : \pi\vert^k \models \varphi_1$ and $\pi\vert^i \models \varphi_2$

Table 1: Syntax and semantics of BLTL. $\pi = \pi^0 \xrightarrow{a_0} \pi^1 \xrightarrow{a_1} \pi^2 \ldots$ is a path. $\pi|^i$ is the suffix of π starting at π^i. $\mathcal{L}$ is given and maps states, π^i, to the subset of atomic propositions that are true in that state.

Some state-of-the-art techniques for the PMC problem in MDPs [1, 18] usually rely on symbolic methods to encode the state-action graph of the MDP in compact representations [9, 10]. Using this representation, such approaches compute the exact maximum probability of satisfying the property through an iterative method that propagates information throughout the state space.

Fully probabilistic models, like Markov chains, exhibit probabilism but not nondeterminism. These models admit only the trivial scheduler that selects the single available distribution at each state. The PMC problem for fully probabilistic systems then reduces to deciding whether the probability of satisfying φ under that scheduler is greater than θ. For solving this problem, there exists an efficient sampling based technique known as Statistical Model Checking (SMC).

SMC comes in two flavours: *hypothesis testing* solves the PMC problem stated above; independent traces of a system are analysed until a meaningful decision can be reached about the hypothesis "probability of satisfaction of φ is smaller than θ". Without going into much detail, a quantity that measures the relative confidence in either of the hypotheses, called the *Bayes factor* (or the likelihood ratio in the case of the SPRT [31]), is dynamically recomputed until enough statistical evidence has been gathered to make a decision. The other kind of SMC is *interval estimation*, where traces are sampled until a probability of satisfaction can be estimated within some confidence interval [20]. This value is then compared against θ. Hypothesis testing is often faster than interval estimation, whereas interval estimation finds the actual probability of satisfying φ. The suitability of either of the techniques, naturally, depends on the specific problem at hand.

In conclusion, since SMC solves the PMC problem statistically on Markov chains, SMC for MDPs reduces to the problem of finding an optimal scheduler for the PMC problem.

3.4 Memoryless Schedulers

It can be shown that, for the PMC problem with unbounded properties, memoryless schedulers are sufficient to achieve the maximal probability [3, 12]. Bounded LTL, does not share this property, i.e. schedulers that maintain historic information may be more powerful than those relying only

on the current state . However, it has been argued in the literature, that memoryless schedulers are more realistic for resolving nondeterminism in some applications, like distributed systems [11].

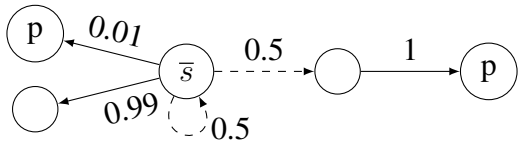

Figure 1: The optimal choice for satisfying the formula $\mathbf{F}^{\leq 2} p$ is to take the dashed transition from the initial state. However, if the system self loops on this choice and takes the dashed transition again, it will necessarily fail to reach p within the time bound. It should follow the solid transition to salvage some probability of doing so. This scheduler has no memoryless equivalent.

In view of this, we restrict the search space to memoryless schedulers, in order to reduce computational cost. For problems that require history-dependent schedulers, we may add a "distance to timeout" variable to AP that effectively allows states to store all relevant historic information. This causes a significant increase in the number of states and, as such, is usually avoided.

4 Statistical Model Checking for MDPs

In this section, we present our algorithm for applying SMC to MDPs. We start with an overview of the procedure and then discuss each of its stages in detail.

4.1 Overview

Up to confidence in the results of classical SMC, the algorithm we propose is a false-biased Monte Carlo algorithm. This means that the algorithm is guaranteed to be correct when it finds a counterexample and it can thus reject $P_{\leq \theta}(\varphi)$. When the algorithm does not manage to find a counterexample, it can retry the search; if it fails once again, then its confidence about the inexistence of such a counterexample becomes higher. In other words, negative answers can always be trusted, and positive answers can eventually be trusted with arbitrarily high confidence. The goal of each run of our algorithm (the flow of which is depicted in Figure 2) is to find a near-optimal scheduler starting from an uninformative uniform candidate.

In an initial *scheduler optimisation stage*, we search for a candidate near-optimal scheduler by iterating over two procedures: the *scheduler evaluation* phase consists in sampling paths from the Markov chain induced by a candidate scheduler σ; using this information, we estimate how likely it is for each choice to lead to the satisfaction of the property φ. The estimates are then used in the *scheduler improvement* phase, in which we update the candidate scheduler σ by *reinforcing* the actions that led to the satisfaction of φ most often. In this way, we obtain a *provably* better scheduler that focuses on the more promising regions of the state space in the next iteration of the scheduler evaluation phase.

In the subsequent *SMC stage*, we use classical SMC (or we could use exact PMC for Markov chains) to check if the candidate scheduler σ from the previous stage settles the original query. If the property $Pr(\{\pi : \pi \in \mathcal{M}^\sigma, \pi \models \varphi\}) \leq \theta$ is false under this scheduler σ, we can safely claim that the MDP does not satisfy the property $P_{\leq\theta}(\varphi)$, because we found a counterexample scheduler σ. Otherwise, we can restart the learning algorithm in an attempt to get a better scheduler. We will show that doing this will exponentially increase confidence in the claim that the MDP satisfies the property.

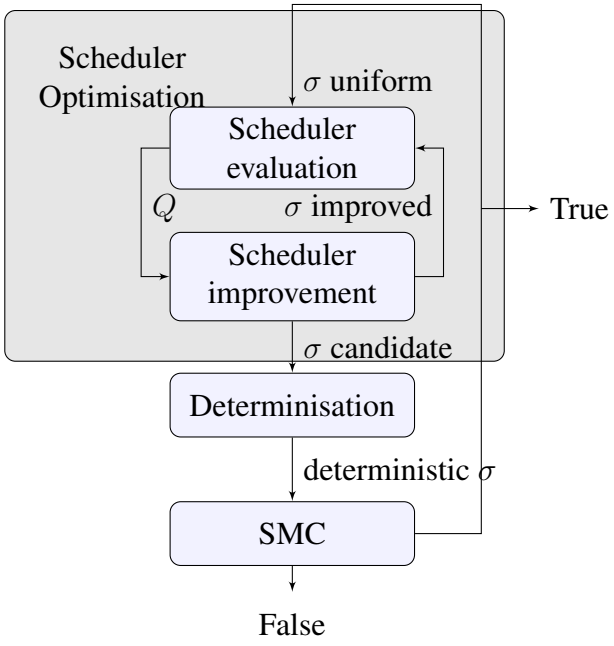

Figure 2: Flowchart of the MDP algorithm

In order to effectively use the sampling information to improve schedulers, we draw from reinforcement learning ideas [27] for choosing near-optimal schedulers in reward maximisation problems. In this setting, it is standard to focus on reinforcing "good" actions based on the immediate rewards they are associated with. These reinforced actions will be preferentially picked by future schedulers. In model checking there is no notion of how good an *individual* action is. Instead, temporal properties induce rewards on whole *paths* rather than on individual actions. Therefore, a path satisfying φ is evidence that the choices generated by the current scheduler along that path are "good". Thus, we reinforce actions that appear in many good paths more than those that appear in few, and modify the scheduler to make them more likely.

4.2 Scheduler Evaluation

Scheduler evaluation is the first of two alternating procedures within the scheduler optimisation stage. It evaluates how good the choices made by a scheduler σ are by repeatedly sampling and checking paths from the Markov chain $\mathcal{M}^\sigma$ induced by σ.

This evaluation checks formula φ on each sampled path π, and *reinforces* each state-action pair in π if $\pi \models \varphi$. In other words, reinforcement is guided by paths, but is applied locally on choices (i.e. on state-action pairs). More formally, for a set of sampled paths $\mathcal{P}$ and state-action pair $(s, a) \in S \times A$, the reinforcements R^+ and R^- are defined as $R^+(s, a) \triangleq |\{\pi \in \mathcal{P} : (s, a) \in \pi \text{ and } \pi \models \varphi\}|$ and $R^-(s, a) \triangleq |\{\pi \in \mathcal{P} : (s, a) \in \pi \text{ and } \pi \not\models \varphi\}|$.

The reinforcement information $R^+(s, a)$ and $R^-(s, a)$ can be used to estimate the probability that a path crossing (s, a) satisfies φ. We denote this probability by $Q(s, a)$, i.e. the *quality* of state-action pair (s, a). As we shall see, a good estimator for $Q(s, a)$ is $\hat{Q}(s, a) = \frac{R^+(s,a)}{R^+(s,a)+R^-(s,a)}$. In the absence of new information from this sampling stage, we leave the quality of (s, a) unchanged. These concepts are formally laid out in Algorithm 1.

Algorithm 1 Scheduler Evaluation

1: **Require:** Scheduler σ, Maximum number of samples N
2: $\forall_{(s,a)\in S\times A} R^+(s, a) \leftarrow 0, R^-(s, a) \leftarrow 0$
3: $\forall_{(s,a)\in S\times A} \hat{Q}^\sigma(s, a) \leftarrow \sigma(s, a)$
4: **for** $i = 1, ..., N$ **do**
5: Sample minimal sufficient path π from $\mathcal{M}^\sigma$
6: **for** $j = 1, ..., |\pi|$ **do**
7: $(s, a) \leftarrow \pi^j$
8: **if** $\pi \models \varphi$ **then**
9: $R^+(s, a) \leftarrow R^+(s, a) + 1$
10: **else**
11: $R^-(s, a) \leftarrow R^-(s, a) + 1$
12: **end if**
13: **end for**
14: **end for**
15: **for** $R^{+/-}(s, a)$ modified in lines 9 or 11 **do**
16: $\hat{Q}^\sigma(s, a) = \frac{R^+(s,a)}{R^+(s,a)+R^-(s,a)}$
17: **end for**
18: **return** $\hat{Q}^\sigma$

Remark 1 (Minimal sufficient paths)**.** *Recall from Subsection 3.2 that, along any path, there is an earliest point where we can decide if the path satisfies φ. After this point, the remainder of the path becomes irrelevant for purposes of deciding about satisfaction or refutation of the property. Thus, we only reward or penalise actions in the minimal sufficient prefix of a path. Any further reward would not be informative.*

4.3 Scheduler Improvement

Scheduler improvement is the second procedure that alternates in the scheduler optimisation stage. It is described in Algorithm 2. It takes as input a scheduler σ and the associated estimated quality function $\hat{Q} : S \times A \to [0, 1]$ from the previous stage. This procedure generates a scheduler σ', an improved version of σ, obtained by greedily assigning higher probability to the most promising actions in each state, i.e. those that led to satisfying φ most often. The remaining probability is distributed according to relative merit amongst all actions. We use a *greediness parameter* $(1 - \epsilon)$ that controls how much probability we assign to the most promising choice. This parameter can be tailored to be small if the system does not require much exploration or large otherwise.

It is important to guarantee that the update does not create a scheduler that blocks the future exploration of any path. If, in the present round, a state-action pair has very poor quality, we want to penalise it, but not disable it entirely. Combining the new greedy choices with the previous scheduler (according to a history parameter h) ensures that no choice is ever blocked as long as the initial scheduler does not block any actions.

Algorithm 2 Scheduler Improvement

1: **Require:** Scheduler σ, History parameter $0 < h < 1$, Greediness parameter $0 < \epsilon < 1$, Quality function estimate $\hat{Q}$
2: $\sigma' \leftarrow \sigma$
3: **for** $s \in S$ **do**
4: $\quad a^* \leftarrow \arg\max_{a \in A} \left\{ \hat{Q}^\sigma(s, a) \right\}$
5: $\quad \forall_{a \in A}\, p(s, a) \leftarrow I\{a = a^*\}(1 - \epsilon) + \epsilon \left(\frac{\hat{Q}^\sigma(s,a)}{\sum_{b \in A} \hat{Q}^\sigma(s,b)} \right)$
6: $\quad \forall_{a \in A}\, \sigma'(s, a) \leftarrow h\sigma(s, a) + (1 - h)p(s, a)$
7: **end for**
8: **return** σ'

4.4 Scheduler Optimisation

Scheduler optimisation simply consists in alternating the scheduler evaluation and the scheduler improvement procedure to incrementally optimise a candidate scheduler.

Since we do not have any prior belief in what constitutes an optimal scheduler, the scheduler is initialised with a Uniform distribution (unbiased!) in each state, ensuring no action is ever blocked. [1] The procedure is described in Algorithm 3.

Remark 2 (Dynamic sampling bounds)**.** *We propose and implement an optimisation to Algorithm 1. If during scheduler evaluation the algorithm has sampled enough satisfying traces to confidently claim that the current scheduler is a counterexample to $P_{\leq\theta}(\varphi)$, then it has answered the original query and may stop sampling. In fact, it may stop learning altogether. Fortunately, Bayesian hypothesis testing provides us with a method to quantify the confidence with which we may answer*

[1] In fact, any probabilistic scheduler that assigns positive probability to all actions would suffice. We choose the uniform scheduler because it maximises entropy.

Algorithm 3 Scheduler Optimisation

Require: σ, h, ϵ, N, Maximum number of alternations between evaluations and improvements L.
for $i = 1, ..., L$ **do**
    $\mathcal{M}^\sigma \leftarrow$ MC induced by MDP $\mathcal{M}$ and scheduler σ
    $\hat{Q} \leftarrow$ SCHEDULEREVALUATE(σ, N)
    $\sigma \leftarrow$ SCHEDULERIMPROVEMENT($\sigma, h, \epsilon, \hat{Q}$)
end for
return σ

a question. Since this method is computationally cheap, it can be used online to stop the algorithm. Alternatively, SPRT [31] could be used to the same end. For further details, please refer to Appendix .1.

4.5 Determinisation

Despite being sufficient to achieve maximum probabilities, deterministic schedulers are a poor choice for exploring the state space through simulation: sampling with a deterministic scheduler provides information *only* for the actions that it chooses. Probabilistic schedulers are more flexible, explore further, and enable reinforcement of different actions. Thus, we always use probabilistic schedulers in the exploration part of our algorithm.

Ideally, σ converges to a near-deterministic scheduler, but due to our commitment to exploration, it will never do so completely. Before using SMC to answer the PMC question, we thus greedily determinise σ. More precisely, we compute a scheduler that always picks the best estimated action at each state. Formally, DETERMINISE(σ) is a new scheduler such that, for all $s \in S$ and $a \in A$

$$\text{DETERMINISE}(\sigma)(s, a) = I\{a = \arg\max_{\alpha \in A(s)} \sigma(s, \alpha)\}$$

We thus hope to redirect the residual probabilities of choosing bad actions to the promising regions of the state space. In practice, this step makes a significant difference.

4.6 Number of Runs

Although we will show that the scheduler optimisation stage converges towards optimal schedulers, at any given point we cannot quantify how close to optimal the candidate scheduler is. Statistical claims are possible, however. If the current candidate is sufficient to settle the original PMC query, the algorithm can stop immediately. If it is not, it may be restarted after a reasonable number of improvement iterations. These restarts help our algorithm finding and focusing on more promising parts of the state space it might have missed before. Algorithms like this are called biased Monte Carlo algorithms. Given a confidence parameter (p) on how likely each run is to converge, we can make a statistical claim up to arbitrary confidence (η) on the number of times we have to iterate the algorithm, $T_{\eta,p}$:

Theorem 1 (Bounding Theorem [5]). *For a false-biased, p-correct Monte Carlo algorithm (with $0 < p < 1$) to achieve a correctness level of $(1 - \eta)$, it is sufficient to run the algorithm at least a number of times:*

$$T_{\eta,p} = \frac{\log_2 \eta}{\log_2(1-p)}$$

This result guarantees that, even in cases where the convergence of the scheduler learning procedure in one iteration is improbable, we will only need to run the procedure a relatively small number of times to achieve much higher confidence. Taking all these considerations into account, the main SMC procedure for MDPs is laid out in Algorithm 4.

Algorithm 4 Statistical Model Checking for Markov Decision Processes

Require: h, ϵ, N, L, Confidence parameter for convergence p, Required confidence η
for $i = 1, ..., T_{\eta,p}$ **do**
    $\forall_{s \in S} \forall_{a \in A(s)}\ \sigma(s,a) \leftarrow \frac{1}{|A|}$
    $\sigma \leftarrow \text{OPTIMISESCHEDULER}(\sigma, h, \epsilon, N, L)$
    $\sigma \leftarrow \text{DETERMINISE}(\sigma)$
    if $\text{HYPOTHESISTESTING}(\mathcal{M}^\sigma, \varphi, \theta)$ = False **then**
        return *False*
    end if
end for
return *Probably True*

An important requirement of this algorithm and Theorem 1 is that we have a positive probability of convergence to an optimal scheduler during scheduler learning. In the next section, we prove this to be the case.

5 Convergence

In this section, we show that the algorithms presented in section 4 are correct. This means that the schedulers found in Algorithm 4 converge to optimal schedulers, under the metric of maximising the probability of satisfying φ.

5.0.1 Scheduler Evaluation

Reinforcement learning algorithms are typically based on estimating quality functions with respect to particular schedulers – functions that quantify how good it is to perform a given action in a given state. In our case, for a property φ, MDP $\mathcal{M}$ and a scheduler σ, the quality function $Q^\sigma : S \times \Sigma \to [0,1]$ associates to each enabled state-action pair (s,a), the probability of satisfying φ, having taken a from s:

$$Q^\sigma(s,a) = \frac{Pr(\{\pi : (s,a) \in \pi, \pi \models \varphi\})}{Pr(\{\pi : (s,a) \in \pi\})}, \tag{1}$$

which is, by definition, the probability of satisfying φ conditioned on having passed through (s, a). Using a common abuse of notation, we will write this expression as

$$Q^\sigma(s, a) = Pr(\pi \models \varphi \mid (s, a) \in \pi) \tag{2}$$

Since our sampling is unbiased, each observation of (s, a) during sampling is an independent, identically distributed estimate of the value of $Q^\sigma(s, a)$. By the Strong Law of Large Numbers, the sequence of empirical averages of these observations converges to the true value of $Q^\sigma(s, a)$ as long as there is a non-zero probability of reaching (s, a) [26]. Furthermore, we know the standard deviation of the error decreases as $1/\sqrt{n}$.

This is enough to guarantee that, with a sufficiently high number of samples, the quality estimation function $\hat{Q}(s, a)$ computed in scheduler evaluation phase (Algorithm 1) approximates the true quality function $Q(s, a)$ arbitrarily well.

5.0.2 Scheduler improvement

In order to analyse scheduler improvement, it will be useful to introduce a quantity related to quality, known as *value*. Value is a measure of how good it is to be in a state for purposes of satisfying φ.

Formally, for a property φ, a MDP $\mathcal{M}$ and a scheduler σ, the value function $V^\sigma : S \to [0, 1]$ associates to each state s the probability of satisfying φ in a path that passes through s:

$$V^\sigma(s) = Pr(\pi \models \varphi \mid (s, a) \in \pi, a \in A(s)\}), \tag{3}$$

Notice that we can compute V^σ from Q^σ by marginalising out the actions enabled at s in Equation 1.

$$V^\sigma(s) = \sum_{a \in A(s)} \sigma(s, a) Q^\sigma(s, a) \tag{4}$$

It is important to notice that for the initial state $\overline{s}$, $V^\sigma(\overline{s}) = Pr(\{\pi : \pi \models \varphi\})$, which is exactly the value we are trying to maximise. We will show that for a scheduler σ' obtained from a scheduler σ in Algorithm 2, $V^{\sigma'}(\overline{s}) \geq V^\sigma(\overline{s})$. Since our goal is to maximise the probability of satisfying φ, this is a guarantee that the algorithm makes progress towards a better scheduler. In order to prove this inequality, we will use a well known theorem from reinforcement learning.

To understand the following results, it is useful to consider the notion of *local update* of a scheduler. Consider two schedulers σ and σ'. The local update of σ by σ' in s, denoted $\sigma[\sigma(s) \mapsto \sigma'(s)]$, is the scheduler obtained by following σ in all states except in state s, where decisions are made by σ' instead. Theorem 2 asserts that, if locally updating σ by σ' always yields a better result than not doing so, then globally updating σ by σ' also yields a better result.

Theorem 2 (Scheduler improvement [27], Section 4.2)**.** *Let σ and σ' be two schedulers and $\forall s \in S : V^{\sigma[\sigma(s) \mapsto \sigma'(s)]}(s) \geq V^\sigma(s)$ then $\forall s \in S : V^{\sigma'}(s) \geq V^\sigma(s)$.*

Proposition 3. *Let σ be the input scheduler and σ' be the output of Algorithm 2. Then $\forall s \in S : V^{\sigma'}(s) \geq V^\sigma(s)$.*

Proof. We first consider the case where the history parameter, h, is 0. In this case $\sigma'(s,a) = I\{a = \arg\max_a Q^\sigma(s,a)\}(1-\epsilon) + \epsilon\left(\frac{Q^\sigma(s,a)}{\sum_{b\in A} Q^\sigma(s,b)}\right)$.

If, for some states s, $Q(s,a) = 0$ for all $a \in A$, then $\sigma(s) = \sigma(s')$ and for such s, the conditions of Theorem 2 are trivially met. Otherwise, define $p_\epsilon(s,a) = \epsilon\frac{Q(s,a)}{\sum_{b\in A(s)} Q(s,b)}$. Notice that $\sum_{a\in A} p_\epsilon(s,a) = \epsilon$ and $\sum_{a\in A}\sigma(s,a) = 1$.

$$
\begin{aligned}
& V^{\sigma[\sigma(s)\mapsto\sigma'(s)]}(s) \\
&= \sum_{a\in A(s)} p_\epsilon(s,a)Q^\sigma(s,a) + (1-\epsilon)\max_{a\in A(s)} Q^\sigma(s,a) && (5)\\
&= \sum_{a\in A(s)} p_\epsilon(s,a)Q^\sigma(s,a) + \\
&\quad \Big(\sum_{a\in A(s)}\sigma(s,a) - \sum_{a\in A(s)} p_\epsilon(s,a)\Big)\max_{a\in A(s)} Q^\sigma(s,a) && (6)\\
&= \sum_{a\in A(s)} p_\epsilon(s,a)Q^\sigma(s,a) + \\
&\quad \sum_{a\in A(s)}[\sigma(s,a) - p_\epsilon(s,a)]\max_{a\in A(s)} Q^\sigma(s,a) \\
&= \sum_{a\in A(s)} p_\epsilon(s,a)Q^\sigma(s,a) + \\
&\quad \sum_{a\in A(s)}\left[(\sigma(s,a) - p_\epsilon(s,a))\max_{a\in A(s)} Q^\sigma(s,a)\right] \\
&\geq \sum_{a\in A(s)} p_\epsilon(s,a)Q^\sigma(s,a) + \\
&\quad \sum_{a\in A}[(\sigma(s,a) - p_\epsilon(s,a))Q^\sigma(s,a)] && (7)\\
&= \sum_{a\in A(s)} p_\epsilon(s,a)Q^\sigma(s,a) + \sum_{a\in A(s)}\sigma(s,a)Q^\sigma(s,a) - \\
&\quad \sum_{a\in A(s)} p_\epsilon(s,a)Q^\sigma(s,a) \\
&= \sum_{a\in A(s)}\sigma(s,a)Q^\sigma(s,a) && (8)\\
&= V^\sigma(s)
\end{aligned}
$$

where the equalities in lines 5 and 8 follow from Equation 4 and the inequality in line 7 comes from the fact that for all $a \in A$, $\max_{a \in A(s)} Q^{\sigma}(s,a) \geq Q^{\sigma}(s,a)$.

In the case where $h \neq 0$, we have

$$\sigma'(s,a) = (1-h)\left[I\{a = \arg\max_{a} Q^{\sigma}(s,a)\}(1-\epsilon) + \right.$$

$$\left. + \epsilon \left(\frac{Q^{\sigma}(s,a)}{\sum_{b \in A} Q^{\sigma}(s,b)} \right) \right] + h\sigma(s,a)$$

We can now repeat the above derivation by multiplying all lines from 5 to 8 by $(1-h)$ and adding $h\sigma(s,a)$ to each of them. All (in)equalities still hold.

Therefore, σ and σ' from Algorithm 2 fulfil the conditions of Theorem 2 and the Corollary holds. □

Proposition 3 is enough to show that each round of scheduler evaluation and scheduler improvement produces a better scheduler for satisfaction of φ.

6 Evaluation

We evaluate our procedure on several well-known benchmarks for the PMC problem. First, we use one easily parametrisable case study to present evidence that the algorithm gives correct answers and then we present systematic comparisons with PRISM [18]. Our implementation extends the PRISM simulation framework for sampling purposes. Because we use the same input language as PRISM, many off-the-shelf models and case studies can be used with our approach[2].

Remark 3 (Reinforcement Heuristics)**.** *Our approach allows us to tune the way in which we compute quality and reinforcement information without destroying guarantees of convergence (under easily enforced conditions) but netting significant speedups in practice. These optimisations range from negatively reinforcing failed paths to reinforcing actions differently based on their estimated quality. A description of these optimisations is beyond the scope of this paper; for further details, please refer to Appendix .2.*

All our benchmarks were run on a 32-core, 2.3GHz machine with 128Gb RAM. The Java Virtual Machine used to run our algorithm was allocated 10Gb for the stack and 10Gb for the heap. Similar amounts of memory were initially allocated to PRISM, but we found that whenever PRISM needed substantial amounts of memory (close to 4Gb), the constraining resource became time and the program timed out regardless of the amount of available memory.

[2] All experimental data such as models, results and scripts can be found at http://www.cs.cmu.edu/jmartins/QEST12.zip. PRISM models can be found at http://www.prismmodelchecker.org/casestudies/index.php

6.1 Parametrisation

Our algorithm's parameters generally affect both runtime and the rate of convergence, with dependence on the MDP's structure. In this section we will outline the methods used to decide values for each parameter.

- History h: high h causes slower convergence, whereas small h makes convergence less likely by making sampling variance a big factor. From a range of tests done over several benchmarks, we found 0.5 to be a good *overall* value by achieving a balanced compromise. To reduce h for specific benchmarks, one can fix the other parameters and reduce it until the algorithm ceases to converge.

- Greediness ϵ: experimentally, the choice of $0 < \epsilon < 1$ influences the convergence of the algorithm. However, the heuristics we use do not allow us to set ϵ explicitly but still guarantee that $0 < \epsilon < 1$ (necessary for convergence). For details, we refer to Appendix .2.

- Threshold θ: θ is provided as part of the PMC query. To understand how the relation of θ to the actual probability treshold affects performance, we present results for different values of θ. These values are chosen by first obtaining p through PRISM and then picking values close to it. In the absence of PRISM results, we gradually increase θ until the property becomes false. Finally, interval estimation can give us hints on good estimates for thresholds.

- Numbers of samples N and iterations L: the main factor in runtime is the total number of samples $N \times L$. A higher N yields more confidence in the reward information R of each iteration. A higher L makes the scheduler improve more often. Increasing L at the cost of N without compromising runtime ($N \times L$ constant) allows the algorithm to focus on interesting regions of the state space earlier. We ran several benchmarks using combinations of N and L resulting in the similar total number of samples, and found that a ratio of around 65:1 $N : L$ was a good overall value. The total number of samples is adapted to the difficulty of the problem. Most benchmarks used $N = 2000$ and $L = 30$, with smaller values possible without sacrificing results. Harder problems sometimes required up to $N = 5000$ and $L = 250$. If the ratio $N : L$ is fixed, N and L are just a bound on runtime. If $\theta > p$, the algorithm will generally run $N \times L$ samples, but if $\theta < p$, it will generally terminate sooner.

- Number of runs T: if a falsifying scheduler is found, the algorithm may stop (up to confidence in SMC). If not, then confidence can be increased as detailed in Section 4.6. We used between 5 and 10 for our benchmarks.

- Statistical Model Checking: the Beta distribution parameters used were $\alpha = \beta = 0.5$ and Bayes factor threshold $T = 1000$. For an explanation these parameters, see [32].

6.2 Correctness, Performance and Implementation

To showcase the correctness and performance of our implementation, we use a simple but easily parametrisable mutex scenario: several processes concurrently try to access a critical region under

# processes	10	20	30	50	100
# states	$\sim 10^4$	$\sim 10^7$	$\sim 10^{10}$	$\sim 10^{15}$	$\sim 10^{31}$
out	0.9825	0.9850	0.9859	0.9850	0.9869
t (s)	98	325	497	1072	6724

Table 2: Mutex results for Algorithm 4 with 10 runs ($T_{\eta,p} = 10$).

a mutual exclusion protocol. All processes run the same algorithm except for one, which has a bug that allows it to unlawfully enter the critical region with some small probability. We want to find the scheduler that makes a mutex violation most likely. We can add processes to increase the state and action space of the system, making it easy to regulate the difficulty of the problem.

Figure 3 demonstrates the behaviour of the learning algorithm (Algorithm 3). The graph plots the ratio of satisfied to unsatisfied sampled traces as the learning algorithm improves the initial scheduler. Although, as expected, learning takes longer for harder problems, the learning trend is evident.

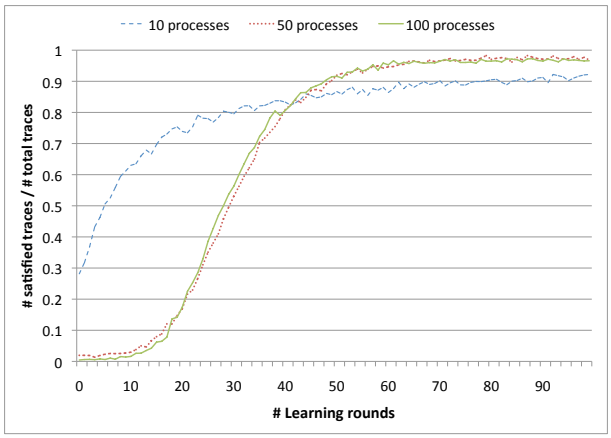

Figure 3: Improvement of schedulers by Algorithm 3

Correctness and performance results for the main algorithm (Algorithm 4) with different parameters are summarised in Table 2. We vary the number of concurrently executing processes in the mutex case study, exponentially increasing the state space. Notice that the time necessary to run Algorithm 4 scales very favourably with the size of the state space. The probability presented is the result of performing interval estimation [32] using the algorithm's most improved scheduler[3]. It is not the estimated probability of satisfying the property with the optimal scheduler. The maximal probability has been computed by exact methods to be 0.988 in all cases.

[3]Although we only present Algorithm 4 with hypothesis testing, interval estimation to produce probability estimates for the best scheduler found.

6.2.1 Parallelisation

One major advantage of using SMC is that sampling is highly and easily parallelisable: all sampling threads have access to the original model and can work independently. This contrasts with the more monolithic approach of exact methods. Consequently, there are significant improvements with multi-threaded operation. However, since the rewards, R^+ and R^-, are shared information, threads have to synchronise their access. This results in diminishing returns.

By going from one thread to ten threads, we reduced runtime to under 25% of its original value and sometimes as low as 17%. Adding up to 20 threads still reduces runtime but only by another 5% of original runtime at best. This points at promising synchronisation reducing optimisations for future work. For these reasons, we used 20 sampling threads.

It is also worth noting that the algorithm itself uses a single thread but is extremely lightweight when compared to sampling. In all benchmarks, checking formulae, rewarding paths and updating schedulers usually account for less than 5% of runtime, and always less than 10%. The remaining time is spent sampling traces. Therefore, faster sampling methods for PRISM or other MDP specifications have the potential to decrease runtime very significantly.

6.3 Comparison and Benchmarks

Statistical approaches are usually very efficient because they only search and store information for a relatively small fraction of the state space. As such, to solve problems that intrinsically require optimal scheduling decisions to be made on *all* states, any statistical method would need to visit the entire state space, thus defeating its own purpose.

Fortunately, a large class of real life problems only require a relatively small set of crucial decisions to be made correctly; symmetry and structure arise naturally in many situations, making some regions of the state space more relevant than others for given properties. This notion of how structured a system is turns out to be an important factor on the performance of our algorithm. In this section, we explore some benchmarks where this phenomenon is evident and thus we divide them in three broad categories:

1. Heavily structured systems: these models are symmetrical by their very nature or because of simplifying assumptions. Common examples are distributed protocols, which have several agents running the exact same algorithm. We present two benchmarks about wireless communication taken from the PRISM benchmark suite.

2. Structured models: these models have some symmetry, but due to noise or irregularity of the environment, not as much as the highly structured systems. Common examples are problems like motion planning or task scheduling by robots. We present a new and comparatively complex motion planning model to illustrate this case.

3. Highly unstructured (random) models: these models have no symmetry whatsoever and exist more as a thought experiment to take the idea of lack of structure to the extreme. We have implemented a random MDP generator for testing these models.

6.3.1 Heavily Structured Systems

Heavily structured systems often have small regions of the state space that regulate the satisfaction of pertinent properties, a feature that our algorithm exploits successfully. In these cases, exact methods also do well, as they can use symbolic approaches that collapse the many symmetries of these models to represent and manipulate the state space with very efficient encodings. Most available benchmarks from the PRISM suite fall under this category.

Since our approach does not represent the state space symbolically, it is not surprising that in several benchmarks of this kind we are outperformed by PRISM. We present only one of these benchmarks as a representative of its class. However, as we move towards more and more complex, unstructured benchmarks, our algorithm starts outperforming traditional methods. In Table 3, we present two case studies: WLAN and CSMA. WLAN models a two-way handshake mechanism of a Wireless LAN standard. We can parametrise a backoff counter. CSMA is a protocol for collision avoidance in networks. We can parametrise the number of agents and a backoff counter. The comparison with PRISM for these two benchmarks is presented in Table 3. Since it is known that hypothesis testing for SMC is much harder when θ is close to the true probability threshold [32], we choose most values of θ close to these thresholds to stress test the approach. Times ($\boldsymbol{t}$) are presented in seconds and are an average of the time spent by 10 different executions of Algorithm 4, each with $T_{\eta,p} = 10$, i.e. 10 restarts until claim of *Probably True*. The middle rows, **out**, show the result of hypothesis testing for the values of θ in the top rows. It is important to notice that an F^* result means that not all executions of the algorithm were able to find the counterexample scheduler.

Notice that for smaller values of θ, the elapsed time is typically shorter because we are allowed to stop as soon as we find a counterexample, whereas when the property is actually satisfied, we have to perform all $T_{\eta,p} = 10$ runs before claiming *Probably True*.

6.3.2 Structured Systems

Structured systems also make fair benchmarks for our algorithm. They still have enough structure to cause some actions to be more important than others but, because of natural irregularity or noise, lack the symmetry that characterises their highly structured counterparts. For this reason, symbolic methods fail to scale in such systems.

We present a new motion planning case study. Each of two robots living in a $n \times n$ grid world must plan a course of action to pick up some object and then meet with the other robot. At each point in time, either robot can try to move 10 grid units in any of the four cardinal directions, but each time a robot moves, it has some probability of scattering and ending up somewhere in a radius of r grid units of the intended destination. Furthermore, robots can only move across safe areas such as corridors or bridges. Table 4 showcases this benchmark with grids of several sizes. Times ($\boldsymbol{t}$) are presented in seconds and are an average of the time spent by 5 different executions of the algorithm, each with $T_{\eta,p} = 10$ runs, i.e. 10 restarts until claim of *Probably True*. The middle rows, **out**, show the result of hypothesis testing for the values of θ in the top rows. For this case study, we use a negative reinforcement heuristic to aggressively avoid unsafe areas. [4]

[4]PRISM has three engines: sparse, hybrid and MTBDD (symbolic). We always compare against the engine that

CSMA 3 4	θ	0.5	0.8	0.85	0.9	0.95	**PRISM**
	out	F	F	F	T	T	0.86
	t	1.7	11.5	35.9	115.7	111.9	136
CSMA 3 6	θ	0.3	0.4	0.45	0.5	0.8	**PRISM**
	out	F	F	F	T	T	0.48
	t	2.5	9.4	18.8	133.9	119.3	2995
CSMA 4 4	θ	0.5	0.7	0.8	0.9	0.95	**PRISM**
	out	F	F	F	F	T	0.93
	t	3.5	3.7	17.5	69.0	232.8	16244
CSMA 4 6	θ	0.5	0.7	0.8	0.9	0.95	**PRISM**
	out	F	F	F	F	F*	timeout
	t	3.7	4.1	4.2	26.2	258.9	timeout
WLAN 5	θ	0.1	0.15	0.2	0.25	0.5	**PRISM**
	out	F	F	T	T	T	0.18
	t	4.9	11.1	124.7	104.7	103.2	1.6
WLAN 6	θ	0.1	0.15	0.2	0.25	0.5	**PRISM**
	out	F	F	T	T	T	0.18
	t	5.0	11.3	127.0	104.9	102.9	1.6

Table 3: Experimental results in several PRISM benchmarks for queries about maximum probability. Times presented in seconds. A * indicates that only some of the executions of the algorithm found a counterexample scheduler.

Robot $n = 50$ $r = 1$	$\boldsymbol{\theta}$	0.9	0.95	0.99	**PRISM**
	out	F	F	F	0.999
	$\boldsymbol{t}$	23.4	27.5	40.8	1252.7
Robot $n = 50$ $r = 2$	$\boldsymbol{\theta}$	0.9	0.95	0.99	**PRISM**
	out	F	F	F	0.999
	$\boldsymbol{t}$	71.7	73.9	250.4	3651.045
Robot $n = 75$ $r = 2$	$\boldsymbol{\theta}$	0.95	0.97	0.99	**PRISM**
	out	F	F	F*	timeout
	$\boldsymbol{t}$	382.5	377.1	2676.9	timeout
Robot $n = 200$ $r = 3$	$\boldsymbol{\theta}$	0.85	0.9	0.95	**PRISM**
	out	F	F	T	timeout
	$\boldsymbol{t}$	903.1	1129.3	2302.8	timeout

Table 4: Experimental results in the motion planning scenario. Times in seconds. A * indicates that only some of the executions of the algorithm found a counterexample scheduler. Checking the formula $P_{\leq\theta}([\mathit{Safe}_1 \mathbf{U}^{\leq 30} (\mathit{pickup}_1 \wedge [\mathit{Safe}'_1 \mathbf{U}^{\leq 30} \mathit{RendezVous}])] \wedge [\mathit{Safe}_2 \mathbf{U}^{\leq 30} (\mathit{pickup}_2 \wedge [\mathit{Safe}'_2 \mathbf{U}^{\leq 30} \mathit{RendezVous}])])$.

As the size of the grid increases, so does the starting distance between the robots and consequently the probability of failure. This is because the scattering effect has more chances to compound and impact the robots' trajectory. Since PRISM failed to return an answer for the last two cases, we have analytically computed an upper bound on the expected probability of satisfying the property. For the case $n = 75, r = 2$, we expect the probability of satisfying the property to be lower than 0.998 and for the case $n = 200, r = 3$ it should be lower than 0.966. These are conservative estimates and the actual probabilities may be smaller than these values, as for example in case $n = 200, r = 3$ with threshold 0.95.

6.3.3 Unstructured (Random) Systems

Completely unstructured systems are particularly difficult for the PMC problem. On one hand, statistical approaches struggle, as no region of the space is more relevant, making directed search ineffective. On the other hand, symbolic approaches cannot exploit symmetry to collapse the state space and also fail to scale.

We implemented an unstructured MDP generator to evaluate performance in these systems for both approaches. Unsurprisingly, exact methods designed to take advantage of symmetry do not scale for these models and provide answers only for the smallest case studies. For large systems, our algorithm also fails to converge quickly enough and after 5 hours (our time bound) the best

performs best in each benchmark, which is the hybrid in this case, and the MTBDD in all others.

schedulers found tipically still only guarantee around 20% probability of success (out of more than 60% actual probability for optimal scheduling for most case studies). The main reason for timeout is the slowdown of the method as larger and larger schedulers need to be kept in memory. Since there is no structure in the system, all regions of the state space are roughly as important as all others and as such, an explicit scheduling function must be built for *all* regions of the state space, which defeats the purpose of approximate methods.

7 Conclusions and Future Work

Combining classical SMC and reinforcement learning techniques, we have proposed what is, to the best of our knowledge, the first algorithm to solve the PMC problem in probabilistic nondeterministic systems by sampling methods.

We have implemented the algorithm within a highly parallel version of the PRISM simulation framework. This allowed us to use the PRISM input language and its benchmarks.

In addition to providing theoretical proofs of convergence and correctness, we have empirically validated the algorithm. Furthermore, we have done extensive comparative benchmarks against PRISM's numerical approach. PRISM managed to outperform our method for the class of very structured models, which a symbolic engine can represent efficiently. For large, less structured systems, our method provided very accurate results for a fraction of the runtime in a number of significant test cases. In fact, the statistical nature of our algorithm enabled it to run, without sacrificing soundness, in benchmarks where PRISM simply failed to provide an answer due to memory or time constraints.

Future challenges for improving the effectiveness of this technique include learning of compositional schedulers for naturally distributed systems, i.e. one scheduler for each agent, and sampling strategies that skip over regions of the state space for which scheduling decisions are already clear.

We did not attempt to optimise PRISM's sampling method. Since sampling accounts for over 90% of our runtime, any increase in sampling performance can have a decisive impact on the efficiency of the implementation. Further technical optimisations are possible by reducing synchronisation requirements and making the implementation fully parallel.

This work is a first step in the statistical verification of probabilistic nondeterministic systems. There are still many interesting possibilities for improving the functionality of the technique. For example, it would be interesting to investigate how to handle schedulers with memory. Another potentially interesting research direction would be adapting the work in [13] and [30] to extend our algorithm to allow the verification of temporal properties without bounds.

References

[1] Christel Baier, Edmund M. Clarke, Vassili Hartonas-Garmhausen, Marta Z. Kwiatkowska, and Mark Ryan. Symbolic model checking for probabilistic processes. In *ICALP*, pages 430–440, 1997.

[2] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artif. Intell.*, 72(1-2):81–138, 1995.

[3] Andrea Bianco and Luca de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. S. Thiagarajan, editor, *FSTTCS*, volume 1026 of *Lecture Notes in Computer Science*, pages 499–513. Springer, 1995.

[4] Jonathan Bogdoll, Luis María Ferrer Fioriti, Arnd Hartmanns, and Holger Hermanns. Partial order methods for statistical model checking and simulation. In Roberto Bruni and Jürgen Dingel, editors, *FMOODS/FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2011.

[5] Gilles Brassard and Paul Bratley. *Algorithmics : theory and practice*. Englewood Cliffs, N.J. Prentice Hall, 1988.

[6] Hyeong Soo Chang, Michael C. Fu, Jiaqiao Hu, Steven, and I. Marcus. A survey of some simulation-based algorithms for markov decision processes. communication in information and systems.

[7] Frank Ciesinski and Christel Baier. Liquor: A tool for qualitative and quantitative linear time analysis of reactive systems. In *QEST*, pages 131–132. IEEE Computer Society, 2006.

[8] Frank Ciesinski and Marcus Größer. On probabilistic computation tree logic. In Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, Joost-Pieter Katoen, and Markus Siegle, editors, *Validation of Stochastic Systems*, volume 2925 of *Lecture Notes in Computer Science*, pages 147–188. Springer, 2004.

[9] Fujita M. McGeer P.O. McMillan K.L. Clarke, E.M. and J.C. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Int. Workshop on Logic Synthesis*, 1993.

[10] Luca de Alfaro, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Roberto Segala. Symbolic model checking of probabilistic processes using mtbdds and the kronecker representation. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 395–410. Springer, 2000.

[11] Sergio Giro. Undecidability results for distributed probabilistic systems. In Marcel Vinicius Medeiros Oliveira and Jim Woodcock, editors, *SBMF*, volume 5902 of *Lecture Notes in Computer Science*, pages 220–235. Springer, 2009.

[12] Tingting Han and Joost-Pieter Katoen. Counterexamples in probabilistic model checking. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 2007.

[13] Ru He, Paul Jennings, Samik Basu, Arka P. Ghosh, and Huaiqing Wu. A bounded statistical approach for model checking of unbounded until properties. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE*, pages 225–234. ACM, 2010.

[14] David Henriques, João Martins, Paolo Zuliani, André Platzer, and Edmund M. Clarke. Statistical model checking for markov decision processes. In *QEST*, pages 84–93. IEEE Computer Society, 2012.

[15] B. Jeannet, P. D'Argenio, and K. Larsen. Rapture: A tool for verifying Markov decision processes. In I. Cerna, editor, *Proc. Tools Day, affiliated to 13th Int. Conf. Concurrency Theory (CONCUR'02)*, Technical Report FIMU-RS-2002-05, Faculty of Informatics, Masaryk University, pages 84–98, 2002.

[16] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[17] Joost-Pieter Katoen, Maneesh Khattri, and Ivan S. Zapreev. A markov reward model checker. In *QEST*, pages 243–244, 2005.

[18] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.

[19] Richard Lassaigne and Sylvain Peyronnet. Approximate verification of probabilistic systems. In Holger Hermanns and Roberto Segala, editors, *PAPM-PROBMIV*, volume 2399 of *Lecture Notes in Computer Science*, pages 213–214. Springer, 2002.

[20] Richard Lassaigne and Sylvain Peyronnet. Probabilistic verification and approximation. *Ann. Pure Appl. Logic*, 152(1-3):122–131, 2008.

[21] Richard Lassaigne and Sylvain Peyronnet. Approximate planning and verification for large markov decision processes. In Sascha Ossowski and Paola Lecca, editors, *SAC*, pages 1314–1319. ACM, 2012.

[22] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[23] Diana El Rabih and Nihal Pekergin. Statistical model checking using perfect simulation. In Zhiming Liu and Anders P. Ravn, editors, *ATVA*, volume 5799 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 2009.

[24] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 202–215. Springer, 2004.

[25] Koushik Sen, Mahesh Viswanathan, and Gul A. Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In *QEST*, pages 251–252, 2005.

[26] Satinder P. Singh and Richard S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1-3):123–158, 1996.

[27] Richard S. Sutton and Andrew G. Barto. Reinforcement learning i: Introduction, 1998.

[28] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *FOCS*, pages 327–338. IEEE Computer Society, 1985.

[29] Håkan L. S. Younes. Ymer: A statistical model checker. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 429–433. Springer, 2005.

[30] Hakan L. S. Younes, Edmund M. Clarke, and Paolo Zuliani. Statistical verification of probabilistic properties with unbounded until. In Jim Davies, Leila Silva, and Adenilso da Silva Simao, editors, *SBMF*, volume 6527 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2010.

[31] Håkan L. S. Younes and Reid G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 223–235. Springer, 2002.

[32] Paolo Zuliani, André Platzer, and Edmund M. Clarke. Bayesian statistical model checking with application to simulink/stateflow verification. In Karl Henrik Johansson and Wang Yi, editors, *HSCC*, pages 243–252. ACM ACM, 2010.

.1 Dynamic Sampling and Bounds

Bayesian hypothesis testing accumulates evidence for and against each hypothesis during sampling. It decides which hypothesis to accept once the evidence in favour of one of the hypothesis reaches a given threshold.

This evidence, called Bayes factor, denoted $\mathcal{B}$, is computed from the original query and the number of satisfying and falsifying samples [32]. One can also understand it as the confidence with which some one of the two hypothesis can be definitely accepted. It thus makes sense to use $\mathcal{B}$ as a stopping criterion for sampling.

The confidence threshold at which a definite decision is made is parametrisable. Since the learning phase is more exploratory, the threshold can be more lax. The SMC phase can be a lot more rigorous by increasing the threshold.

We know that hypothesis testing becomes much harder if θ approaches the real probability. This becomes a problem if the algorithm stops learning exactly when the candidate scheduler achieves near-θ probability. To avoid this, the dynamic bounds are calculated for confidence in answering the $P_{\leq\theta+\iota}(\varphi)$ query, where ι is a relatively small probability increment (0.05 by default). In this situation, our algorithm can achieve a scheduler above the required θ, which makes the more rigorous $P_{\leq\theta}(\varphi)$ much easier, and faster, to answer.

.2 Alternative Reinforcement

We allow the specification of different reward and scheduler update schemes under very mild restrictions. As long as we start with a probabilistic scheduler that assigns positive probability to each action and a scheduler update scheme that uses a history parameter $h > 0$, we are guaranteed to preserve some exploration ability. After we have run a few aggressive learning rounds with the unorthodox scheduler update scheme, we can revert to the usual update scheme and still guarantee overall convergence. If the aggressive learning is enough to solve the initial query, of course, we can stop the procedure prematurely as usual.

Negative last step reinforcement is a heuristic particularly suited for problems with some concept of "safety constraints", i.e. regions of the state space where the system must remain. In order to converge to a near-optimal scheduler in fewer iterations, we may wish to very quickly avoid the decisions that made a path cross from the safe region into the unsafe region.

The motion planning case study is a good example for negative reinforcement schemes. It makes sense to assign a very large penalty (big negative reinforcement) to the actions that make the robot leave the safe region. As a result, the robots wander around aimlessly at first, but quickly learn which actions make them hit the walls or fall off the bridge. This allows them to more easily converge towards better schedulers.

On the other hand, in other problems where satisfaction of the BLTL formula is hard, even taking correct actions may have only a very small probability of success. In these cases, the few positive reinforcements a "good" action would get are overwhelmed by the negative reinforcements from all the unsatisfying paths. Negative reinforcement becomes a source of noise. The takeaway is that there is no best heuristic, but the flexibility to use them allows us to tackle much larger problems.

.3 Action Accountability

In this section, we detail a formalisation of what we mean by "action responsible for failure" in situations where safety regions exist.

It is a tedious explanation of a subtle concept and we largely believe it to be unnecessary and cumbersome to the understanding of the paper. We present it, if the reviewers wish, for the sake of completeness, to review this process.

We will eschew a bit of formalism to motivate the problem at first, before presenting the formal framework.

As can be seen by the negative reinforcement heuristics from the motion planning case study, explained in Appendix .2, identifying the action which was responsible for falsifying a property can be critical.

There is a default way to assign *action accountability* to BLTL. This particular notion of accountability will not always match the one we desire since it turns out that a good notion of accountability is inherent to problem under study.

To illustrate, recall that the property with which each robot must comply in our motion planning example is of the form $\varphi = S_1 \mathbf{U}^{\leq 30} \left(\mathit{Chk} \wedge \left[S_2 \mathbf{U}^{\leq 30} G \right] \right)$, where S_1 and S_2 are safety zones, *Chk* is a *checkpoint*, and G is the final goal. The property expresses that the robot should move within the S_1 safe region until it reaches the checkpoint, then move within the S_2 safe region until it reaches the goal. Given this intuition, it should be clear that an action responsible for falsifying the property will belong to one of the following categories 1) violates S_1, 2) reaches *Chk* and then violates S_2, or 3) reaches *Chk* but fails to reach G within the time bounds.

Also notice that a naïve way to check this property (as any Until property) is to get a path, look at the first state and say "Is the right side of the Until ($S_2 \mathbf{U}^{\leq 30} G$) true in this state? If it is, I am done and the formula is true. Otherwise, let me check if the left side of the Until (S_1) is true in this state. If it is not, I am done and the formula is false. Otherwise I ask the same question again in the next state."

Suppose that the robot is trying to reach *Chk*, but turns away from it and hits a wall. In this case, S_1 is violated and the last action, the one that made the robot hit the wall, is penalised. This is correct.

Now suppose that we have a path where the robot has kept within the safe area S_1, reached the checkpoint *Chk*, took some steps in S_2 but failed to remain there. Of course, we wish the accountable action to be the one that violated S_2. However, if we are checking φ naïveli, we will move along S_1 until we reach the checkpoint. At this point, we will ask "Is the right side of the Until ($S_2 \mathbf{U}^{\leq 30} G$) true in this state?". Since it is not, and here lies the rub, instead of holding the failure in S_2 accountable, we will press on saying "Otherwise, let me check if the left side of the Until (S_1) is true in this state". Since we (correctly) left S_1 to be in S_2 instead, we will naïvely conclude that moving out of S_1 to S_2 was responsible for violating the property!

What we actually want is for the *accountability* to transfer to $\left[S_2 \mathbf{U}^{\leq 30} G \right]$ as soon as *Chk* is satisfied.

We must then define a framework for specifying where the accountability lies. To this end, we propose using a sequence of partial functions, called *accountability functions* $A : \mathbb{N} \times \Pi \times \mathcal{F} \mapsto \mathbb{N}$, where Π is the set of all finite paths, $\mathcal{F}$ is the set of all BLTL formulae. $A^k_\pi(\varphi)$ returns the position

in $\pi \in \Pi$ of the state-action pair that accounts for the satisfaction of $\varphi \in \mathcal{F}$, in the suffix path $\pi|^k$. To find the accountable action, a call to $A^k_\pi(\varphi)$ applies the first function in the sequence whose domain includes φ. We enforce that one rule *always* has to apply, i.e. the sequence is exhaustive in $\mathcal{F}$.

To make things clearer, for the naïve semantics, accountability is given by the following rules. The burden of checking a proposition is when evaluated: $A^k_\pi(p) = k$. Negation simply passes the burden to its inner formula: $A^k_\pi(\neg\varphi) = A^k_\pi(\varphi)$. The $\wedge$ connective is a little more involved. For $\varphi = \varphi_1 \wedge \varphi_2$,

$$A^k_\pi(\varphi) = \begin{cases} \max\{A^k_\pi(\varphi_1), A^k_\pi(\varphi_2)\} & \pi|^k \models \varphi_1, \pi|^k \models \varphi_2 \\ A^k_\pi(\varphi_1) & \pi|^k \not\models \varphi_1, \pi|^k \models \varphi_2 \\ A^k_\pi(\varphi_2) & \pi|^k \models \varphi_1, \pi|^k \not\models \varphi_2 \\ \min\{A^k_\pi(\varphi_1), A^k_\pi(\varphi_2)\} & \pi|^k \not\models \varphi_1, \pi|^k \not\models \varphi_2 \end{cases}$$

Intuitively, the accountability for $\varphi_1 \wedge \varphi_2$, if both are true, is given by whichever took longer to check. When one of φ_1, φ_2 is false, the burden is on whichever side failed to be satisfied. Finally, if both are true, the burden is on the first to fail. Alternatively, it could have been on φ_1 exclusively, as it happens, for instance, in programming languages: φ_2 is never actually checked if φ_1 is false. The critical function is the one for $\varphi = \varphi_1 \mathbf{U}^{\leq n} \varphi_2$.

$$A^k_\pi(\varphi) = \begin{cases} A^k_\pi(\varphi_2) & \pi|^k \models \varphi_2 \\ A^{k+1}_\pi(\varphi_1 \mathbf{U}^{\leq n-1} \varphi_2) & \pi|^k \models \varphi_1, \pi|^k \not\models \varphi_2, n > 0 \\ k & \text{otherwise} \end{cases}$$

So, the above functions, in any order, provide the default accountability semantics in the implementation. For the motion planning case study, we wish to create a special rule to deal with checkpoints. Therefore, anywhere before the rule for $\varphi_1 \mathbf{U}^{\leq n} \varphi_2$, we add the following rule for $\varphi = S_1 \mathbf{U}^{\leq n_1} \big(\mathit{Chk} \wedge \big(S_2 \mathbf{U}^{\leq n_2} G\big)\big)$.

$A^k_\pi(\varphi) =$

- $A^k_\pi(S_2 \mathbf{U}^{\leq n_2} G)$ when $\pi|^k \models S_1, \pi|^k \models \mathit{Chk}, n > 0$
- $A^{k+1}_\pi\left(S_1 \mathbf{U}^{\leq n_1 - 1}\big(\mathit{Chk} \wedge \big(S_2 \mathbf{U}^{\leq n_2} G\big)\big)\right)$ when $\pi|^k \models S_1, \pi|^k \not\models \mathit{Chk}, n > 0$
- k otherwise

By applying this function before the one that deals with arbitrary **U**'s, we achieve the desired semantics. The first case finds the checkpoint and assigns accountability entirely to the inner **U** as soon as the checkpoint is reached. Furthermore, since rules are recursive, it handles arbitrarily many checkpoints.