# A Simple and Efficient Statistical Model Checking Algorithm to Evaluate Markov Decision Processes

Benoît Delahaye, Axel Legay, and Sean Sedwards

INRIA/IRISA, France, {benoit.delahaye,axel.legay, sean.sedwards}@inria.fr

**Abstract.** We propose a simple and efficient technique that allows the application of statistical model checking (SMC) to Markov Decision Processes (MDP). Our technique finds schedulers that transform the original MDP into a purely stochastic Markov Chain, on which standard SMC can be used. A statistical search is performed over the set of possible schedulers to find the best and worst with respect to the given property. If a scheduler is found that disproves the property, a counter-example is produced. If no counter-example is found, the algorithm concludes that the property is *probably* satisfied, with a confidence depending on the number of schedulers evaluated.

Unlike previous approaches, the efficiency of our algorithm does not depend on structural properties of the MDP. Moreover, we have devised an efficient procedure to address general classes of schedulers by using Pseudo Random Number Generators and Hash Functions. In practice, our algorithm allows the representation of general schedulers in constant space, in contrast to existing algorithms that are exponential in the size of the system. In particular, this allows our SMC algorithm for MDPs to consider memory-dependant schedulers in addition to the memoryless schedulers that have been considered by others.

## 1 Introduction

Markov decision processes are a convenient model to represent systems containing both probabilistic and non-deterministic transitions, where the non-determinism may represent behaviour that is unspecified or deployment-dependent. In such a model it is often required to find the maximum or minimum probability that it will satisfy a particular property, considering all possible schedulers of a given class.

*Problem* Given an MDP and property $\varphi$, verify (i) that there exists a deterministic scheduler s.t. $\mathrm{P}(\varphi) \geq \theta$ or (ii) that there does not exist a scheduler s.t. $\mathrm{P}(\varphi) \leq \theta$. In both cases the procedure is to find a scheduler that maximises $\mathrm{P}(\varphi)$, but such schedulers may be arbitrarily rare.

*Basic idea* Iterate a randomised procedure $T$ times that has non-zero probability $p$ of finding the scheduler that maximises $\mathrm{P}(\varphi)$. If the outcome of each iteration is independent of previous iterations, the algorithm will either find a scheduler s.t. $\mathrm{P}(\varphi) \geq \theta$ within arbitrary user-defined confidence bounds or, with confidence $\propto (1-p)^T$, conclude that no scheduler exists. In case (i) the algorithm is *p-correct*, *true-biased*; in case (ii) the algorithm is *p-correct*, *false-biased*.

## 2 Related Work

*Clarke algorithm* The following procedure is repeated $T$ times: refine a probabilistic scheduler $L$ times, starting from a neutral scheduler, to become more focused around the scheduler that maximises $\mathrm{P}(\varphi)$; determinise the final probabilistic scheduler to estimate $\mathrm{P}(\varphi)$ using SMC.

The refinement is achieved as follows. For each of the $L$ steps, generate $N$ independently random samples from the joint distribution formed by probabilistic schedulers and traces. Count the occurrences of scheduler transitions ((*state*,*action*) pairs) in the samples that satisfy $\varphi$ and thus produce the next probabilistic scheduler by increasing the probability of such 'good' transitions in proportion to the their occurrence.

With $N$ infinite, $p = 1$ and a single refinement step is all that is needed to find a probabilistic scheduler that can be determinised to maximise $\mathrm{P}(\varphi)$. The effect of finite $N$, s.t. $p < 1$, is counteracted by making $L > 1$, $T > 1$ and introducing a heuristic *history* parameter. The history parameter defines the fraction of results to carry forward from the previous refinement step, to avoid losing good transitions that were not seen in the current $N$ simulations. Increasing the history parameter allows a greater coverage of the distribution and thus increases $p$.

A further heuristic parameter, *greediness*, speeds up convergence to a deterministic scheduler by effectively determinising some of the scheduling during the refinement. By reducing some of the variance associated with the scheduling, it potentially allows more accurate coverage of the probabilistic distribution, but risks converging to a sub-optimal scheduler and thus reducing $p$.

*Allocating resources between $T$, $L$ and $N$* The confidence of the result is increased by increasing $T$ or $p$. $p$ is increased by increasing $L$ and $N$. For a given problem, how should a fixed computational budget be allocated?

Refinement is an optimistic strategy that simulations will tend to be confined to a region of scheduler×trace space that can be adequately covered by simulation. This may not be justified, even in the case of well structured models, especially when the traces are long. The advantage of the refinement approach is that successive iterations increasingly focus on better schedulers that avoid unproductive areas of the scheduler space. The disadvantage is that such focusing may only find a local optimum when the initial $N$ simulations do not adequately represent the distribution. By iterating the entire refinement process $T$ times, the algorithm implicitly expects fewer than $T$ local maxima. It is not clear why, in general, $L > 1$ will be better than performing all simulations in a single step.

With standard Monte Carlo estimation of a Bernoulli parameter, it is not necessary to see a significant fraction of the support; the error depends only on the number of samples in relation to the parameter. By contrast, refinement requires $N$ to be sufficiently large to adequately cover the joint distribution of schedulers and traces. With increasingly long traces, the size of scheduler space increases exponentially with respect to the number of sampled states that attempt to cover it. The majority of (*state*,*action*) pairs may therefore only be seen once. The algorithm thus does not entirely escape from the 'state explosion problem'.

$T$ is related to confidence $\eta$ and $p$ by

$$T = \frac{\log \eta}{\log(1-p)},$$

for $\log$ to any base. Hence

$$\frac{\mathrm{d}T}{\mathrm{d}p} = \frac{\ln \eta}{(1-p)\left(\ln(1-p)\right)^2}$$

Assuming a required confidence of $\eta = 0.95$, $\left|\frac{\mathrm{d}T}{\mathrm{d}p}\right| \approx 1$ when $p = 0.22$. $\left|\frac{\mathrm{d}T}{\mathrm{d}p}\right| > 1$ when $p < 0.22$, so for equal computational effort it is better to increase $p$ to improve confidence. When $p > 0.22$ it is better to just increase $T$.

# 3 Background

## 3.1 Schedulers and MDPs

Markov Decision Processes are a common formalism for modelling discrete state transition systems exhibiting both nondeterministic and probabilistic behaviours. In this section, we recall background on MDPs and Schedulers.

**Definition 1 (Markov Decision Process).** *A Markov decision Process (MDP) is a tuple $\mathcal{M} = \langle S, s^0, A, T, \mathcal{L}\rangle$ where $S$ is a finite set of states, $s^0 \in S$ is an initial state, $A$ is a finite set of actions, $T : S \times A \times S \mapsto [0,1]$ is a transition function such that for all $s \in S, a \in A$, either $\sum_{s' \in S} T(s,a,s') = 1$ ($a$ is enabled) or $\sum_{s' \in S} T(s,a,s') = 0$ ($a$ is disabled) and for all $s$ there exists at least one action $a$ that is enabled, and $\mathcal{L} : S \to 2^{AP}$ is a labelling function mapping each state to the set of atomic propositions true in that state.*

A *Markov Chain* is an MDP where the transition function is fully probabilistic. In order to allow probabilistic choice between available actions, we relax the definition of MDPs by replacing the assumption "for all $s \in S, a \in A$, either $\sum_{s' \in S} T(s,a,s') = 1$ or $\sum_{s' \in S} T(s,a,s') = 0$" with "$\sum_{a \in A} \sum_{s' \in S} T(s,a,s') = 1$". A Path in MDP $\mathcal{M} = \langle S, s^0, A, T, \mathcal{L}\rangle$ is an execution of the form $s_0 \xrightarrow{a_1} s_1 \dots \xrightarrow{a_n} s_n$ such that for all $0 \leq i \leq n-1$, $a_{i+1}$ is enabled in $s_i$ and for all $1 \leq j \leq n$, $T(s_{i-1}, a_i, s_i) > 0$. The set of paths of $\mathcal{M}$ is denoted $\mathit{\Pi}_{\mathcal{M}}$. The set of finite paths of $\mathcal{M}$ is denoted $\mathit{\Pi}^*_{\mathcal{M}}$.

A scheduler for a MDP resolves nondeterminism after each sequence of states $s_0, \dots s_n \in S^+$ by providing a distribution over the set of actions enabled in $s_n$ as the next step.

**Definition 2 (Scheduler).** *A* scheduler *for a MDP $\mathcal{M} = \langle S, s^0, A, T, \mathcal{L}\rangle$ is a function $\sigma : S^+ \times A \to [0,1]$ such that for all finite sequence of states $w = s_0, \dots s_n \in S^+$, we have $\sigma(w,a) > 0$ only if $a$ is enabled in $s_n$ and $\sum_{a \in A} \sigma(w,a) = 1$.*

The are several classes of schedulers that are addressed in practice. *Deterministic* schedulers are schedulers for which either $\sigma(w,a) = 1$ or $\sigma(w,a) = 0$ for all pairs $(w,a) \in S^+ \times A$. Effectively, deterministic schedulers are reduced to functions $\sigma : S^+ \to A$. *Memoryless* schedulers are schedulers for which decisions are only based on the last state of each sequence, i.e. such that for all $a \in A$ and for all sequence $w = s_0, \dots s_n$ and $w' = s'_0, \dots s'_k$ with $s_n = s'_k$, we have $\sigma(w,a) = \sigma(w',a)$. Effectively, memoryless schedulers are reduced to functions $\sigma_m : S \times A \to [0,1]$.

Given a MDP $\mathcal{M}$, any scheduler $\sigma$ induces a (potentially infinite-state) Markov chain by resolving all nondeterministic choices. This Markov chain is denoted $\mathcal{M}^\sigma$.

**Definition 3 (Markov Chain induced by a scheduler).** *Given a MDP* $\mathcal{M} = \langle S, s_0, A, T, \mathcal{L}, \rangle$ *and a scheduler* $\sigma : S^+ \times A \to [0,1]$ *for* $\mathcal{M}$*, the (infinite-state)* Markov chain *induced by* $\sigma$ *is* $\mathcal{M}^\sigma = \langle S^+, w_0, A, T^*, \mathcal{L}^* \rangle$*, where* $w_0 = s_0$ *is the finite sequence of length* $1$ *consisting of only* $s_0$*,* $\mathcal{L}^*$ *is such that for all finite sequence* $w$ *ending in* $s_n$ $\mathcal{L}^*(w) = \mathcal{L}(s_n)$*, and for all* $w$*,* $T^*(w,a,w') = \sigma(w,a)T(s_n,a,s_{n+1})$ *if* $w$ *ends in* $s_n$ *and* $w' = w.s_{n+1}$*, and* $0$ *otherwise.*

### 3.2 Probabilistic and Statistical Model Checking

We are interested in verifying that a given MDP $\mathcal{M}$ satisfies a formula $\varphi$ specified in Bounded Linear Temporal Logic (BLTL) [3] with a probability at most (resp. at least) $\theta$. This problem, denoted $\mathbb{P}_{\leq\theta}(\varphi)$ (resp. $\mathbb{P}_{\geq\theta}(\varphi)$), consists in deciding whether, for all schedulers $\sigma$ for $\mathcal{M}$, the measure of the set of traces in the induced Markov chain $\mathcal{M}^\sigma$ is greater than (resp. lower than) $\theta$. Exact Probabilistic Model Checking techniques exist to solve this problem [1, 4, 6]. They rely on symbolic techniques to encode the MDP in efficient data structures and numerical methods to compute the exact maximum (resp. minimum) probability of satisfying the given property.

Another approach to the problem is to reduce it to finding the scheduler $\sigma_m$ that maximizes (resp. minimizes) the measure of the set of traces satisfying $\varphi$, and compare the measure of this set *under scheduler* $\sigma_m$ to $\theta$. It has been proven [2] (Remark 10.99) that *deterministic* schedulers are sufficient for achieving maximum probability, hence we only consider *deterministic* schedulers from this point. In [5], Henriques et al. propose to use an adaptation of Kearns' learning algorithm to find a locally optimal scheduler $\sigma$ and then to use Statistical Model Checking (SMC) on $\mathcal{M}^\sigma$ to check satisfaction of $\mathbb{P}_{\leq\theta}(\varphi)$.

Statistical Model checking [7] comes in two flavours: *hypothesis testing*, that directly solves $\mathbb{P}_{\leq\theta}(\varphi)$ without computing the actual probability $\mathbb{P}(\varphi)$; and *interval estimation*, that estimates the probability $\mathbb{P}(\varphi)$ using samples of traces and then compares it to $\theta$. In both cases, the result is obtained with confidence and precision parameters that we denote $\mathsf{conf}$ and $\mathsf{pre}$ respectively.

In the rest of the paper, we propose to sample from deterministic schedulers and check each induced Markov chain using SMC techniques. If a scheduler disproving the property is found, a counterexample is produced. After $N$ positive trials, we conclude that the property is *probably* satisfied with confidence $\mathsf{conf} = \dots$ **Sean, can you fill-in?**. Our technique is detailed in the next section.

## 4 Statistical Model Checking MDPs

In this section, we present in details our SMC algorithm for evaluating MDPs. We first introduce and explain the algorithm, and then identify the challenges that such an algorithm raises and study its complexity and convergence bounds.

### 4.1 Main Algorithm

Our SMC algorithm for MDPs is based on the following observation: In learning algorithms such as the ones presented in [5], a lot of effort is spent on optimizing *probabilistic* schedulers in order to derive the optimal *deterministic* scheduler. This technique is complex and implies a huge memory consumption, as one has to store quality measures for each state-action pair. Furthermore, the technique does not scale in the case of unstructured systems because the learning algorithm needs some structure to make progress. Finally, due to the large memory consumption, the authors have to limit themselves to memoryless schedulers. We choose to follow an orthogonal approach based on statistical sampling of *deterministic* schedulers to search for the optimum. While our technique does not learn or optimize schedulers, it is much less memory-intensive and behaves in the same way for structured and unstructured systems. All the effort spent on optimizing in the case of learning algorithms is here spent on searching for new random deterministic schedulers. By the law of large numbers, the optimal scheduler will be found eventually and a lower bound on the confidence in our algorithm can be computed. Finally, once this algorithm is coupled with a very efficient technique for representing schedulers in practice (which we present in Section 5), our SMC algorithm for MDPs becomes very efficient in terms of memory consumption.

```
Data: MDP $\mathcal{M}$, Property $\varphi$, Probability bound $\theta$, SMC Confidence conf, SMC Precision
      pre, Number of rounds $N$
Result: probablytrue, (false, scheduler)
for int $i = 1$, $i \leq N$, $i{+}{+}$ do
    Deterministic Scheduler $\sigma_d$ = Random();
    if !SMC($\mathcal{M}^{\sigma_d}$, $\varphi$, $\theta$, conf, pre) then
        Return (false, $\sigma_d$);
Return probablytrue;
```

Algorithm 1: SMC for MDP

The SMC algorithm for MDPs, presented in Algorithm 1, is simplistic in theory: Given a number $N$ as a parameter, it samples $N$ deterministic schedulers $\sigma_d$ and performs SMC on the induced Markov chain $\mathcal{M}^{\sigma_d}$. As soon as a scheduler is found that falsifies the property, the algorithm returns false and the counter-example $\sigma_d$. If no such scheduler is found, the algorithm returns probablytrue, meaning that the property is probably satisfied, with a confidence that depends on $N$. Although very simple, Algorithm 1 raises several challenges that need to be resolved in order to provide efficient solutions. The main challenge is to be able to represent deterministic schedulers efficiently, and to sample traces from the resulting Markov chain. By definition, a deterministic scheduler $\sigma_d$ is a function that maps finite histories $w \in S^+$ to choices of available actions. Representing general deterministic schedulers in practice would thus require infinite memory. As a consequence, researchers usually restric themselves to memoryless schedulers, which can be represented with a linear amount of memory in the size of the system. Although memoryless schedulers are sufficient for computing maximal and minimal bounds for infinite properties [], they are not sufficient for bounded properties []. In the following section, we present solutions that allow to represent wider classes of deterministic schedulers in constant memory, by using hash functions and random number generators. These methods are coupled with efficient algorithms that allow sampling from the resulting Markov chain.

### 4.2 Convergence and Bounds

Since our algorithm is based on random sampling from the set of general schedulers, the law of large numbers ensures that the optimal scheduler will eventually be found. Moreover, since we are looking for counter-example schedulers to the property under test $\mathbb{P}_{\leq\theta}\varphi$, the absolute optimal scheduler may not need to be found: a *good enough* scheduler may suffice as long as it allows us to disprove the property. This algorithm thus amounts to a false-biased Monte Carlo algorithm. Such algorithms have been widely studied in the litterature [], and confidence bounds exist, which we report here. Let $p$ be the measure of probability of optimal schedulers, which may be very small. By [**?**], the required number of samples $N_{p,\eta}$ to achieve a confidence bound of $1-\eta$, where $\eta$ is the probability of answering probablytrue while the correct answer is false, is

$$N_{p,\eta} = \frac{\log(\eta)}{\log(1-p)}.$$

For instance, if $\eta = .01$ and $p = 10^{-5}$, then $N_{p,\eta} \equiv 460514$.

## 5 Schedulers

As seen in the previous section, one of the main challenges of performing SMC for MDPs is to be able to represent schedulers in an efficient way, and to simulate the Markov Chain induced by a given scheduler.

In this section, we propose several algorithms that allow to sample the Markov chain induced by $\mathcal{M}$ under a given scheduler $\sigma$. The most important feature of these algorithms is that the internal representation of scheduler $\sigma$ is an integer. Selecting uniformly among schedulers is thus reduced to choosing uniformly a random integer. The section is structured as follows: we first introduce briefly the notions of (pseudo)random number generators and hash functions, on which our algorithms are base; We then introduce our algorithms, starting from the less expressive one which allows to understand the underlying concepts easily albeit addressing a restricted class of schedulers, and then increase expressivity until we address the class of general schedulers.

### 5.1 Random Number Generators and Hash Functions

*(Pseudo)Random Number Generators* A *Random Number Generator (RNG)* is a function that allows to produce an infinite sequence of random numbers that are evenly distributed on a given interval (parameter of the function). In practice, RNGs allow to resolve nondeterminism in a uniform way, e.g. allowing to choose one transition among a set of available transitions. Unfortunately, real RNGs cannot be adressed in practice. *Pseudorandom Number Generators (PRNGs)* have been developed for this purpose. A PRNG is an algorithm that generates an infinite sequence of numbers that approximate the properties of a truly random sequence. The sequence is not truly random as it is entirely determined using a small set of parameters. In practice, PRNGs are initialized using a seed $\xi$, e.g. in the form of an integer, that is then used to generate the sequence of pseudo random numbers. Once the seed is fixed, the sequence will always be the same. Since true RNGs cannot be computed, PRNGs are often used in computation to approximate uniform distributions. In the rest of the document, PRNGs will be adressed as $\mathsf{PR}_\xi : \mathbb{N} \to [0, 1]$, such that $\mathsf{PR}_\xi(n)$ returns the $n$th number of the sequence initialized with seed $\xi$.

*Hash Functions* A *Hash Function* is an algorithm that allows to map a large set of data to a smaller set of fixed size. In our case, hash functions will be used to map vectors $\mathbf{v}$ of integers or reals to a single integer or real number. The main characteristics of hash functions are their uniformity: the images should be uniformly distributed on the target set. One additional property that we will require in our algorithms is that hash functions are randomly distributed: there should be no correlation between the images of two vectors that are themselves correlated. A perfect hash function will ensure that there are no *collisions* in it image, i.e. that two distinct vectors cannot have the same image. However, due to physical limitations, real hash functions will not be able to ensure this property, and their aim will be to minimize the number of collisions.

### 5.2 Using PRNG to resolve nondeterminism

When simulating sample executions of a system that resolves nondeterministic choices with a uniform probabilistic decision (i.e., using the *uniform scheduler*), it is usual to make such decisions using a PRNG. In our case, we want to select a deterministic scheduler uniformly and use it to resolve nondeterminism in the MDP of interest. Choosing a scheduler amounts to deciding *in advance* how all nondeterministic choices will be resolved.

*Example 1.* Consider the MDP $\mathcal{M}_{\mathsf{ex}}$ given in Figure 1. The initial state is $s_0$ and all states have nondeterministic choices between actions $a_1$ and $a_2$, each corresponding transition leading to distributions on next states. The overall execution tree of $\mathcal{M}_{\mathsf{ex}}$ up to depth 3 is given in Figure 2. Choosing a scheduler for $\mathcal{M}_{\mathsf{ex}}$ amounts to choosing a subtree where all nondeterministic choices are resolved. Such subtrees are given later in Figures 4,6, and 8.

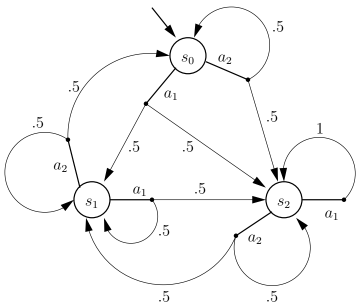

Fig. 1: MDP $\mathcal{M}_{\mathsf{ex}}$

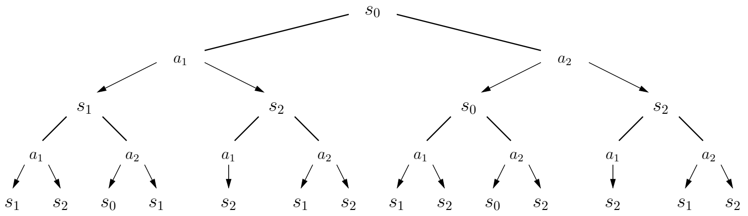

Fig. 2: Partial execution tree of $\mathcal{M}_{\mathsf{ex}}$

In practice, deciding in advance how all nondeterministic choices are resolved may be very costly, as it implies storing potentially infinite data. If PRNGs are used in order to resolve nondeterministic choices *on the fly*, then storing the seed that initializes a given PRNG is sufficient to be able to reproduce all choices identically. Since fixing the seed of a PRNG suffices to determine the sequence of random numbers generated, storing a PRNG scheduler amounts to storing its seed. In the following, we propose a simplistic algorithm that allows to simulate the Markov Chain induced by a deterministic scheduler $\sigma_{\xi}^{R}$, generated using a PRNG initialized with seed $\xi$. Although we will see later on that this algorithm only allows to represent an unrealistic subclass of schedulers, it is needed for understanding. In the following, the function $\mathsf{resolve}(r, A)$ returns an element $a$ of set $A$ chosen uniformly using random number $r$, and the function $\mathsf{Nextstate}_{\mathcal{M}}(s, a)$ resolves the probabilistic choices according to the distribution associated to action $a$ in state $s$.

As expected, the above algorithm produces simulations of the Markov chain induced by MDP $\mathcal{M}$ and scheduler $\sigma_{\xi}^{R}$. Indeed, given a finite sequence $w = s_0, \dots s_n$,

```
Data: MDP $\mathcal{M}$, seed $\xi$, length $L$
Result: Trace $w$ of $\mathcal{M}^{\sigma_\xi^R}$ of length $L$
State $s = \mathcal{M}$.initialstate;
Trace $w = s$;
for int $i = 1$, $i \leq L$, $i$++ do
    Action $a = \mathsf{resolve}(\mathsf{PR}_\xi(i), s.\text{Actions})$;
    $s = \mathsf{Nextstate}_\mathcal{M}(s, a)$;
    $w$.append($s$);
Return $w$;
```

Fig. 3: PRNG Scheduler

the choice produced by $\sigma_\xi^R$ for the next action will always be the same, determined by $\mathsf{resolve}(\mathsf{PR}_\xi(n), s_n.\text{Actions})$. The main advantage of Algorithm 3 is that it allows to represent deterministic schedulers by only storing the seed $\xi$ used for initializing the PRNG. Moreover, since every nondeterministic choice is taken uniformly among the available actions using independent random numbers, selecting a seed $\xi$ uniformly allows to select uniformly among all schedulers of this form.

However, this algorithm only allows to represent a non-realistic subclass of schedulers. Indeed, we remark that the scheduler $\sigma_\xi^R$ induced by this algorithm is such that all nondeterministic choices taken at depth $n$ depend on the same random number $\mathsf{PR}_\xi(n)$. In practice, this means that whenever $w = s_0, \dots s_n$ and $w' = s'_0, \dots s'_n$ are such that $s_n$.Actions = $s'_n$.Actions, we will have $\sigma_\xi^R(w) = \sigma_\xi^R(w')$. In the following, we propose more involved algorithms that allow to target larger classes of schedulers.

*Example 2 (PRNG Scheduler Algorithm).* Consider MDP $\mathcal{M}_{\mathsf{ex}}$ depicted in Figure 1. Applying the Algorithm 3 to $\mathcal{M}_{\mathsf{ex}}$ with a given seed $\xi$ yields the Markov chain $\mathcal{M}_{\mathsf{ex}}^{\sigma_\xi^R}$ depicted as a darker subtree of the executions of $\mathcal{M}_{\mathsf{ex}}$ in Figure 4. Notice that choices of the associated scheduler $\sigma_\xi^R$ depend on the depth of the execution. Indeed, nondeterministic choices labeled $\mathbf{2}$ and $\mathbf{3}$ are resolved using the same random number $\mathsf{PR}_\xi(2)$, and thus select the same action $a_1$. Similarly, the resolution of nondeterministic choices labelled $\mathbf{4}$, $\mathbf{5}$ and $\mathbf{6}$ will be resolved using the same random number $\mathsf{PR}_\xi(3)$. As a consequence, a scheduler resembling the one depicted in Figure 6 cannot be obtained using the Algorithm 3, since choices labelled $\mathbf{2}$ and $\mathbf{3}$ in this scheduler are resolved in a different way.

### 5.3 Advanced scheduling using PRNG and Hash Functions

We now propose more advanced algorithms that use the same principle as Algorithm 3 for the representations of the scheduler, but use PRNG together with hash functions in

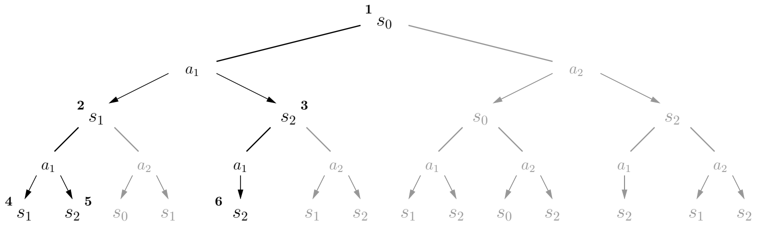

Fig. 4: Execution of $\mathcal{M}_{\mathsf{ex}}^{\sigma_\xi^R}$ according to the Algorithm 3

order to simulate the underlying Markov chain. Using both PRNG and hash functions allows to address larger classes of schedulers, as we show below.

*Memoryless Schedulers* A state of the system is an assignment of values to a vector of $n$ system variables $v_i$. Each $v_i$ is represented by a number of bits $b_i$, typically corresponding to a primitive data type. The state vector can thus be represented by an integer of $\sum_{i=1}^{n} b_i$ bits, denoted $\mathbf{v}$. Our algorithm works by generating a pseudo-random number that is dependent on the state of the system at any given instant and the chosen scheduler. The scheduler is represented by an integer $\xi$, that our algorithm chooses uniformly at random. We thus generate a hash code $h_\xi(\mathbf{v})$ from scheduler seed $\xi$ and state vector $\mathbf{v}$. $h$ is then used as the seed of a standard linear congruential PRNG, that is used to make the next non-deterministic choice in the simulation.

```
Data: MDP $\mathcal{M}$, seed $\xi$, length $L$, hash function $\mathsf{Hash}$
Result: Trace $w$ of $\mathcal{M}^{\sigma_\xi^H}$ of length $L$
State $s = \mathcal{M}$.initialstate;
Trace $w = s$;
for int $i = 1$, $i \leq L$, $i$++ do
    Vector $\mathbf{v} = s$.vector;
    Int $h_\xi = \mathsf{Hash}(\xi, \mathbf{v})$;
    Action $a = \mathsf{resolve}(\mathsf{PR}_{h_\xi}(1), s.\text{Actions})$;
    $s = \mathsf{Nextstate}_{\mathcal{M}}(s, a)$;
    $w$.append($s$);
Return $w$;
```

Fig. 5: H-PRNG Scheduler Algorithm

The choice of the hash function used in our algorithms is of paramount importance, as we need to be able to range over all schedulers. In the rest of the section, we assume we have a *perfect* hash function $\mathsf{Hash}$ in order to present our algorithms. We discuss the right choice for $\mathsf{Hash}$ in practice in Section **??**.

The H-PRNG scheduler algorithm mixes the use of the hash function $\mathsf{Hash}$ introduced above with random numbers for resolving nondeterministic choices at each step. The resulting scheduler is denoted $\sigma_\xi^H$. As expected, Algorithm 5 produces traces of $\mathcal{M}^{\sigma_\xi^H}$, and $\sigma_\xi^H$ is a correct scheduler. Indeed, for all sequence $w = s_0, \dots s_n$ of $\mathcal{M}$, the choice produced by $\sigma_\xi^H$ for the next action will always be the same, determined by $\mathsf{resolve}(\mathsf{PR}_{h_\xi(s)}(1), s.\text{Actions})$, where $h_\xi(s) = \mathsf{Hash}(\xi, \mathbf{v})$. As for Algorithm 3, Algorithm 5 allows to use a single integer value (the seed $\xi$) in order to represent the scheduler $\sigma_\xi^H$. Since $\mathsf{Hash}$ is a perfect hash function, selecting a seed $\xi$ uniformly allows to select uniformly among all schedulers that can be otained this way. Contrary to Algorithm 3, Algorithm 5 allows to represent a meaningfull class of schedulers: *memoryless schedulers*. Indeed, the resolution of nondeterministic choices in Algorithm 5 is only based on the original seed $\xi$ and on the vector representing the last state visited $\mathbf{v}$. Thus, for a given scheduler $\sigma_\xi^H$, the seed $\xi$ is fixed and the resolution only depends on the vector representing the last state visited $\mathbf{v}$. This is enough to justify that all schedulers of the type $\sigma_\xi^H$ are memoryless. Moreover, if the hash function is well chosen and there are no collisions, all local choices of the type $\mathsf{resolve}(\mathsf{PR}_{h_\xi(s)}(1), s.\text{Actions})$ are independent as they use the same PRNG $\mathsf{PR}$ initialized with a different seed. By the properties of PRNGs and hash functions, the following property holds: for all memoryless scheduler $\sigma$, there exists a seed $\xi$ such that for all state $s$, we have $\sigma(s) = \mathsf{resolve}(\mathsf{PR}_{h_\xi(s)}(1), s.\text{Actions})$, which implies that $\sigma = \sigma_\xi^H$.

*Example 3 (H-PRNG Scheduler Algorithm).* Consider MDP $\mathcal{M}_{\mathsf{ex}}$ depicted in Figure 1. Applying Algorithm 5 to $\mathcal{M}_{\mathsf{ex}}$ with a given seed $\xi$ yields the Markov chain $\mathcal{M}_{\mathsf{ex}}^{\sigma_\xi^H}$ depicted as a darker subtree of the executions of $\mathcal{M}_{\mathsf{ex}}$ in Figure 6. Notice that the scheduler $\sigma_\xi^H$ is memoryless, i.e. resolution of nondeterministic choices only depend on the current state in the execution. Indeed, nondeterministic choices labeled $\mathbf{1}$ and $\mathbf{2}$ are resolved using the same random number $\mathsf{PR}_{h_\xi(s_0)}(1)$, and thus select the same action $a_2$. Similarly, the resolution of nondeterministic choice labeled $\mathbf{4}$ will also be $a_2$ and nondeterministic choices labeled $\mathbf{3}, \mathbf{5}$ and $\mathbf{6}$ will be resolved using the same random number $\mathsf{PR}_{h_\xi(s_2)}(1)$. As a consequence, a scheduler resembling the one depicted in Figure 8 cannot be obtained using Algorithm 5, since choices labelled $\mathbf{1}$ and $\mathbf{2}$ in this scheduler are resolved in a different way.

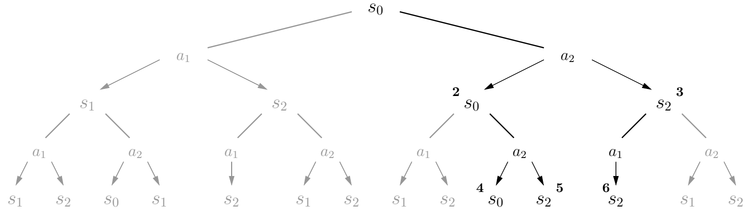
$s_0$ 1
$a_1$
$a_2$
$s_1$
$s_2$
2 $s_0$
$s_2$ 3
$a_1$ $a_2$ $a_1$ $a_2$ $a_1$ $a_2$ $a_1$ $a_2$
$s_1$ $s_2$ $s_0$ $s_1$ $s_2$ $s_1$ $s_2$ $s_1$ $s_2$ 4 $s_0$ $s_2$ 5 6 $s_2$ $s_1$ $s_2$

Fig. 6: Execution of $\mathcal{M}_{\mathsf{ex}}^{\sigma_\xi^H}$ according to Algorithm 5

The above result assumes that there are no collisions in the hash function. In practice, this may pose problems as collisions might appear. If this is the case, then some schedulers may not be representable. However, we assume that the number of collisions is limited, and thus that deterministic memoryless schedulers that cannot be represented using Algorithm 5 are rare (neglectable). In the following, we present other algorithms that allow to represent even larger classes of schedulers.

```
Data: MDP $\mathcal{M}$, seed $\xi$, length $L$, hash function Hash
Result: Trace $w$ of $\mathcal{M}^{\sigma_\xi^D}$ of length $L$
State $s = \mathcal{M}$.initialstate;
Trace $w = s$;
for int $i = 1$, $i \leq L$, $i$++ do
    Vector $\mathbf{v} = s$.vector;
    Int $h_\xi^i = \mathsf{Hash}(\mathsf{PR}_\xi(i), \mathbf{v})$;
    Action $a = \mathsf{resolve}(\mathsf{PR}_{h_\xi^i}(1), s$.Actions);
    $s = \mathsf{Nextstate}_{\mathcal{M}}(s, a)$;
    $w$.append($s$);
Return $w$;
```

Fig. 7: State-depth Dependant Scheduler Algorithm

*State-depth dependant schedulers* In order to obtain a larger class of schedulers, one can tweek Algorithm 5 in order to use new seeds each time a state is visited along an execution. In Algorithm 5, the scheduler seed $\xi$ is mixed with the state-vector $\mathbf{v}$ using a hash function, and the result is used in order to resolve nondeterminism. A simple way to extend this principle is to use the next random number generated using seed $\xi$ instead of $\xi$ itself: instead of using $h_\xi(\mathbf{v}) = \mathsf{Hash}(\xi, \mathbf{v})$ as a seed for $\mathsf{PR}$, we propose to use $h_\xi^n(\mathbf{v}) = \mathsf{Hash}(\mathsf{PR}_\xi(n), \mathbf{v})$, where $n$ is the current depth in the execution. The resulting scheduler is addressed as $\sigma_\xi^D$.

Although the schedulers defined as above are more general than memoryless schedulers, they still do not encompass all memory-dependant schedulers. Indeed, every scheduler of the form $\sigma_\xi^D$ as defined above suffers from a certain type of dependency: for all sequence $w = s_0, \ldots s_n$ and $w' = s'_0, \ldots s'_n$ with $s'_n = s'_n$, we have $h_\xi^n(w) = h_\xi^n(w')$, i.e. the same hash number is generated. Hence, the resolution of nondeterministic choices after $w$ and $w'$ are not independent. As a consequence, all schedulers cannot be addressed using the State-depth Dependant algorithm presented above. Nevertheless, the set of schedulers we address with this algorithm contains, but is not limited to, memoryless schedulers.

*Example 4 (State-depth Dependant Scheduler Algorithm).* Consider MDP $\mathcal{M}_{\mathsf{ex}}$ depicted in Figure 1. Applying the State-depth dependant scheduler algorithm to $\mathcal{M}_{\mathsf{ex}}$ with a given seed $\xi$ yields the Markov chain $\mathcal{M}_{\mathsf{ex}}^{\sigma_\xi^D}$ depicted as a darker subtree of the executions of $\mathcal{M}_{\mathsf{ex}}$ in Figure 8. Notice that the scheduler $\sigma_\xi^D$ cannot be obtained using Algorithms 3 or 5. However, schedulers obtained using the state-depth dependant algorithm are still not fully general. Indeed, nondeterministic choices labelled **4** and **6** will be resolved using the same random number $\mathsf{PR}_{h_\xi^3(s_1)}(1)$. Similarly, the resolution of nondeterministic choices labelled **5** and **7** will be resolved using the same random number $\mathsf{PR}_{h_\xi^3(s_2)}(1)$.

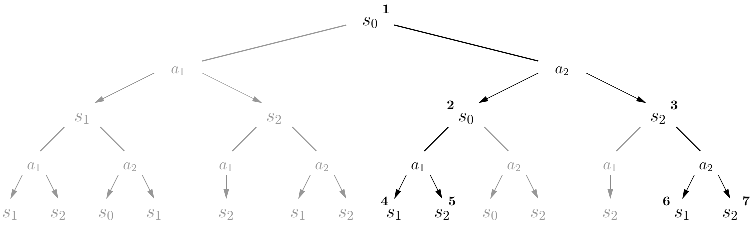

Fig. 8: Execution of $\mathcal{M}_{\mathsf{ex}}^{\sigma_\xi^D}$ according to the State-depth Dependant Scheduler Algorithm

*General schedulers in theory* Following the same principles as in Algorithm 5, one can device an algorithm that addresses memory-dependant schedulers. In Algorithm 5, a random number is generated in each step in order to resolve nondeterminism, using a hash of the seed $\xi$ and the current state vector $\mathbf{v}$. In order to represent memory-dependant schedulers, we propose to use a number $h$ that represents both the current path and the original seed, that we then hash with the state-vector in order to produce a random number that resolves nondeterminism for the next transition. Formally, considering a seed $\xi$, we propose to use the following sequence of numbers $(h_\xi(n))_{n\in\mathbb{N}}$ to produce random numbers that are successively used in order to resolve nondeterminism along an execution. Obviously, each number $h_\xi(n)$ that resolves nondeterminism in the $n^{th}$ step will depend on all choices made up to depth $n$. $h_\xi(n)$ is thus defined recursively as follows. Given seed $\xi$ and initial state $s_0$, we define $h_\xi(0) = \mathsf{Hash}(\xi, \mathbf{v}_0)$, where $\mathbf{v}_0$ is the state-vector corresponding to state $s_0$. When resolving nondeterminism in the $n^{\mathsf{th}}$ step after a history $w = s_0, \ldots, s_n$, we define $h_\xi(n) = \mathsf{Hash}(h_\xi(n-1), \mathbf{v}_n)$, where $\mathbf{v}_n$ is the state-vector corresponding to state $s_n$. Such a scheduler is denoted $\sigma_\xi$.

Obviously, Algorithm 9 indeed yields samples of executions from $\mathcal{M}^{\sigma_\xi}$, and $\sigma_\xi$ is a correct scheduler: after a given history $w = s_0, \ldots s_n$, the next nondeterministic choice is always resolved using the same random number $\mathsf{PR}_{h_\xi(n)}(1)$. Moreover, $\sigma_\xi$ is clearly history-dependant: two different executions $w = s_0, \ldots s_n$ and $w' = s'_0, \ldots s'_k$ ending in the same state $s'_k = s_n$ will resolve nondeterminism in the next state using independant random numbers, generated using $h_\xi(n)$ and $h'_\xi(k)$. Indeed, we have $h_\xi(n) = \mathsf{Hash}(h_\xi(n-1), \mathbf{v}_n)$ and $h'_\xi(k) = \mathsf{Hash}(h'_\xi(k-1), \mathbf{v}_k)$. Although $\mathbf{v}_n = \mathbf{v}_k$,

```
**Data**: MDP $\mathcal{M}$, seed $\xi$, length $L$, hash function $\mathsf{Hash}$
**Result**: Trace $w$ of $\mathcal{M}^{\sigma_\xi}$ of length $L$
State $s = \mathcal{M}$.initialstate;
Trace $w = s$;
Int $h_\xi = \xi$;
**for** *int* $i = 1$, $i \leq L$, $i$++ **do**
    Vector $\mathbf{v} = s$.vector;
    $h_\xi = \mathsf{Hash}(h_\xi, \mathbf{v})$;
    Action $a = \mathsf{resolve}(\mathsf{PR}_{h_\xi}(1), s$.Actions);
    $s = \mathsf{Nextstate}_{\mathcal{M}}(s, a)$;
    $w$.append($s$);
**Return** $w$;
```

Fig. 9: General Scheduler Algorithm

we know that $h_\xi(n-1)$ is independant from $h'_\xi(k-1)$, which ensures independance in the results given that the hash function Hash is perfect. Unfortunately, for this same reason, it appears that Algorithm 9 cannot yield memoryless schedulers. Thus, in order to consider the set of all possible schedulers, Algorithms 5 and 9 will both have to be used in practice.

## References

1. C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Z. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In *ICALP*, vol. 1256 of *Lecture Notes in Computer Science*, pp. 430–440. Springer, 1997.
2. C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.
3. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
4. L. de Alfaro, M. Z. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of probabilistic processes using mtbdds and the kronecker representation. In *TACAS*, vol. 1785 of *Lecture Notes in Computer Science*, pp. 395–410. Springer, 2000.
5. D. Henriques, J. Martins, P. Zuliani, A. Platzer, and E. M. Clarke. Statistical model checking for markov decision processes. In *QEST*, pp. 84–93. IEEE Computer Society, 2012.
6. M. Z. Kwiatkowska, G. Norman, and D. Parker. Prism 4.0: Verification of probabilistic real-time systems. In *CAV*, vol. 6806 of *Lecture Notes in Computer Science*, pp. 585–591. Springer, 2011.
7. A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In *RV*, vol. 6418 of *Lecture Notes in Computer Science*, pp. 122–135. Springer, 2010.