Quantitative Model Checking*

R. Grosu and S.A. Smolka

Department of Computer Science
State University of New York at Stony Brook
grosu,sas@cs.sunysb.edu

Abstract. We present QMC, a one-sided error Monte Carlo decision procedure for the LTL model-checking problem $S \models \varphi$. Besides serving as a randomized algorithm for LTL model checking, QMC delivers *quantitative* information about the likelihood that $S \models \varphi$. In particular, given a specification S of a finite-state system, an LTL formula φ, and parameters ϵ and δ, QMC performs random sampling to compute an estimate $\widetilde{p}_Z$ of the expectation p_Z that the language $L(B)$ of the Büchi automaton $B = B_S \times B_{\neg\varphi}$ is empty; B is such that $L(B) = \emptyset$ iff $S \models \varphi$. A random sample in our case is a lasso, i.e. an initialized random walk through B ending in a cycle. The estimate $\widetilde{p}_Z$ output by QMC is an (ϵ, δ)-approximation of p_Z—one that is within a factor of $1 \pm \epsilon$ with probability at least $1 - \delta$—and is computed using a number of samples N that is optimal to within a constant factor, in expected time $O(N \cdot D)$ and expected space $O(D)$, where D is B's recurrence diameter. Experimental results demonstrate that QMC is fast, memory-efficient, and scales extremely well.

1 Introduction

Model checking [6, 22], the problem of deciding whether or not a property specified in temporal logic holds of a system specification, has gained wide acceptance within the hardware and protocol verification communities, and is witnessing increasing application in the domain of software verification. The beauty of this technique is that when the state space of the system under investigation is finite-state, model checking may proceed in a fully automatic, push-button fashion. Moreover, should the system fail to satisfy the formula, a counter-example trace leading the user to the error state is produced.

Model checking, however, is not without its drawbacks, the most prominent being *state explosion*. This phenomenon can render model checking intractable for many applications of practical interest; see e.g. [**?**], where it is shown that the problem is PSPACE-complete for LTL. Over the past two decades, a number of techniques have been developed to combat state explosion: symbolic model checking, partial-order reduction, symmetry reduction, and bounded model checking. See [5] for a comprehensive discourse on model checking.

We present in this paper an alternative approach to coping with state explosion based on the technique of *Monte Carlo estimation*. Monte Carlo methods are often used in engineering and computer-science applications to compute an approximation of a solution whose exact computation proves intractable, being, for example, NP-hard. Example applications include belief updating in Bayesian networks [9], computing the volume of convex bodies [10], and approximating the number of solutions of a DNF formula [17].

* R. Grosu was partially supported by the NSF Faculty Early Career Development Award CCR01-33583.

Our approach makes use of the following idea from the automata-theoretic technique of Vardi and Wolper [28] for LTL model checking: given a specification S of a finite-state system and an LTL formula φ, $S \models \varphi$ (S models φ) if and only if the language $L(B)$ of the Büchi automaton $B = B_S \times B_{\neg\varphi}$ is empty. Here B_S is the Büchi automaton representing S's state transition graph, and $B_{\neg\varphi}$ is the Büchi automaton for the negation of φ. The presence in B of an accepting lasso—a reachable cycle containing a final state—means that S is *not* a model of φ. Hence, such a lasso can be viewed as a *counter-example* to $S \models \varphi$.

To decide if $L(B)$ is empty, we have developed the `QMC` Monte Carlo approximation algorithm for *quantitative model-checking*. Underlying the execution of `QMC` is a Bernoulli random variable Z that takes value 1 with probability p_Z and value 0 with probability $q_Z = 1 - p_Z$. Intuitively, p_Z is the expectation that $L(B) = \emptyset$ and, in fact, $p_Z = 1$ iff $L(B) = \emptyset$. In general, however, p_Z is unknown and difficult to compute. Thus, `QMC` seeks to estimate p_Z `QMC` by taking N random samples Z_i from B, and uses the average $\widetilde{p}_Z$ of the outcomes as the estimate. A random sample in our case is an initialized random walk in B terminating in a cycle, i.e., a lasso. Such a random walk is constructed *on-the-fly* to avoid the *a' priori* construction of B, which would immediately lead to state explosion. A sample $Z_i = 1$ if its associated lasso is non-accepting, and $Z_i = 0$ otherwise.

To determine $\widetilde{p}_Z$ and, concomitantly, N, `QMC` appeals to the `OAA` optimal approximation algorithm of Dagum et al. [8]. `OAA` computes what is known as an (ϵ, δ)-approximation: one that is within a factor of $1 \pm \epsilon$ with probability at least $1 - \delta$. Through its reliance on `OAA`, the number of samples taken by `QMC` is guaranteed to be optimal to within a constant factor. Now, in performing its random sampling, should `QMC` encounter a $Z_i = 0$ with associated accepting lasso l, it returns false with l as a witness (counter-example). Otherwise, it returns true with error margin ϵ and confidence ratio δ.

Running `QMC` in the just-described manner yields a one-sided error Monte Carlo *decision procedure* for LTL model checking that takes $O(4\ln(2/\delta)/\epsilon)$ samples. `QMC` can also be run in *estimation mode* where rather than terminating and deciding false upon encountering an accepting lasso, it continues sampling as dictated by `OAA` to compute $\widetilde{p}_Z$. As explained in Section 4, `QMC` may not terminate in estimation mode if the number of non-accepting lassos in B is less than a certain quantity. The main features of `QMC` are the following.

- `QMC` performs random sampling of lassos in the Büchi automaton $B = B_S \times B_{\neg\varphi}$ to yield a one-sided error Monte Carlo decision procedure for the LTL model-checking problem $S \models \varphi$.
- Unlike other model checkers,[1] `QMC` also delivers *quantitative* information—in the form of an (ϵ, δ)-approximation of p_Z—about the likelihood that an arbitrary run of a system satisfies a given formula. This has allowed us to observe, for example, that the expectation that a run of a system of n dining

[1] We are referring here strictly to model checkers in the classical sense, i.e., those for nondeterministic/concurrent systems and temporal logics such as LTL, CTL, and the mu-calculus. Model checkers for probabilistic systems and logics, a topic discussed in Section 7, also produce quantitative results.

philosophers is deadlock-free increases linearly with n, an observation that is fairly obvious in retrospect, but to our knowledge has not been reported previously in the literature.
- `QMC` is very efficient in both time and space. Its time complexity is $O(N \cdot D)$ and its space complexity is $O(D)$, where D is B's recurrence diameter. Moreover, by virtue of its reliance on the `OAA` algorithm of [8], the number of samples N taken by `QMC` is optimal to within a constant factor.
- Although we present `QMC` in the context of the classical model-checking problem for nondeterministic/concurrent systems, the algorithm works with little modification on systems specified using stochastic modeling formalisms such as discrete-time Markov chains.
- We have implemented `QMC` in the context of the jMocha model checker for Reactive Modules [1]. A feature of the implementation is that the "next state" along a random walk in search of an accepting lasso is generated by randomly selecting both one of the guarded commands in a nondeterministic choice construct and a valuation for the input variables.
- Our experimental results demonstrate `QMC` is fast, memory-efficient, and scales extremely well. It consistently outperforms jMocha's LTL enumerative model checker, which uses a form of partial-order reduction.

The rest of the paper develops along the following lines. Section 2 reviews LTL model checking. Section 3 provides an overview of the optimal Monte-Carlo estimation algorithm of [8]. Section 4 presents `QMC`, our Monte Carlo model-checking algorithm. Section 5 describes our jMocha implementation of `QMC`. Section 6 summarizes our experimental results. Section 7 discusses previous approaches to randomized model checking. Section 8 contains our conclusions.

2 LTL Model Checking

Given a concurrent system S and temporal-logic formula φ, the *model-checking problem* is to decide whether S satisfies φ. In case φ is a *linear temporal logic* (LTL) formula, the problem can be elegantly solved by reducing it to the *language emptiness problem* for finite automata over infinite words [28]. The reduction involves modeling S and $\neg\varphi$ as Büchi automata B_S and $B_{\neg\varphi}$, respectively, taking the product $B = B_S \times B_{\neg\varphi}$, and checking whether the language $L(B)$ of B is empty.[2]

The set of well-formed LTL formulas (wffs) is constructed from a finite set of atomic propositions AP, the standard boolean connectives, and the temporal operators "neXt state" (`X`) and "Until" (`U`).

Definition 1 (Syntax of LTL formulas). *A* well-formed LTL formula over AP *is defined inductively as follows:*

1. *Every $p \in AP$ is an LTL wff.*

[2] The rationale behind this reduction is as follows: $S \models \varphi$ iff $L(B_S) \subseteq L(B_\varphi)$ iff $L(B_S) \cap \overline{L(B_\varphi)} = \emptyset$ iff $L(B_S) \cap L(B_{\neg\varphi}) = \emptyset$ iff $L(B_S \times B_{\neg\varphi}) = \emptyset$

2. *If φ and ψ are LTL wffs, then so are $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\mathtt{X}\,\varphi$, $\varphi\,\mathtt{U}\,\psi$.*

An interpretation for an LTL formula is an infinite word $\xi = x_0 x_1 \ldots$ over the alphabet $\mathcal{P}(AP)$, i.e., a mapping from the naturals to $\mathcal{P}(AP)$. We write ξ_i for the suffix of ξ starting at x_i.

Definition 2 (Semantics of LTL formulas). *Let ξ be an infinite word over $\mathcal{P}(AP)$. We define the satisfaction relation $\xi \models \varphi$ inductively as follows:*

1. $\xi \models p$ **iff** *$p \in x_0$ for $p \in AP$*
2. $\xi \models \neg\varphi$ **iff** *not $\xi \models \varphi$*
3. $\xi \models \varphi \wedge \psi$ **iff** *$\xi \models \varphi$ and $\xi \models \psi$*
4. $\xi \models \varphi \vee \psi$ **iff** *$\xi \models \varphi$ or $\xi \models \psi$*
5. $\xi \models \mathtt{X}\,\varphi$ **iff** *$\xi_1 \models \varphi$*
6. $\xi \models \varphi\,\mathtt{U}\,\psi$ **iff** *there is an $i \geq 0$ s.t. $\xi_i \models \psi$ and $\xi_j \models \varphi$ for all $0 \leq j < i$.*

A *Büchi automaton* is a finite automaton over infinite words.

Definition 3 (Büchi automaton). *Let Σ be a finite set. A* Büchi automaton *B over Σ is a five-tuple $B = (\Sigma, Q, Q_0, \delta, F)$ where:*

1. *Σ is the* input alphabet.
2. *Q is a finite set of* states.
3. *$Q_0 \subseteq Q$ is the set of* initial states.
4. *$\delta \subseteq Q \times \Sigma \times Q$ is the* transition relation.
5. *$F \subseteq Q$ is the set of* accepting states.

Let $\xi = x_0 x_1 \ldots$ be an infinite word in Σ^ω. A *run* of B over ξ is a mapping $\sigma = s_0 s_1 \ldots$ from the naturals to Q such that $s_0 \in Q_0$ and for all i, $(s_i, x_i, s_{i+1}) \in \delta$. We shall sometimes write a run σ over ξ as $s_0\, x_0\, s_1\, x_1 \ldots$ and refer to it simply as a "run". A *finite run* is a finite prefix of a run. Let $inf(\sigma)$ be the set of states that appear infinitely often in a run σ over ξ. Then, σ is *accepting* if $inf(\sigma) \cap F \neq \emptyset$. The *language* $L(B)$ of B is the set of all infinite words $\xi \in \Sigma^\omega$ having accepting runs in B.

Every LTL formula φ can be translated to a Büchi automaton whose language is the set of infinite words satisfying φ by using the LTL *tableau construction* of [11]. Due to space constraints we omit the construction.

Definition 4 (Product of Büchi automata). *Let $B_1 = (\Sigma, Q_1, Q_1^0, \delta_1, F_1)$ and $B_2 = (\Sigma, Q_2, Q_2^0, \delta_2, F_2)$ be two Büchi automata. The* product Büchi automaton *$B_1 \times B_2 = (\Sigma, Q, Q^0, \delta, F)$ is defined as follows:*

1. $Q = Q_1 \times Q_2 \times \{0,1,2\}, \quad Q^0 = Q_1^0 \times Q_2^0 \times \{0\}, \quad F = Q_1 \times Q_2 \times \{2\}$,
2. $((s_1, s_2, x), \alpha, (t_1, t_2, y)) \in \delta$ **iff** *$(s_1, \alpha, t_1) \in \delta_1$ and $(s_2, \alpha, t_2) \in \delta_2$ and*

   if $x = 0$ *and* $t_1 \in F_1$ **then** $y = 1$
   if $x = 1$ *and* $t_2 \in F_2$ **then** $y = 2$
   if $x = 2$ **then** $y = 0$
   otherwise $y = x$

The x,y in the definition of δ ensure that accepting states of both B_1 and B_2 occur infinitely many times in an accepting run of $B_1 \times B_2$ even though they may never occur simultaneously.

Checking (non-)emptiness of $L(B)$ is equivalent to finding a strongly connected component of B that is reachable from an initial state and contains an accepting state. Due to the acceptance condition for Büchi automata, however, this reduces to finding a reachable accepting *cycle*. Looking for such a cycle is usually done by using the *double depth-first search* algorithm `DDFS` [7, 16] shown below, where `init(B)` = $\mathtt{Q}_0$, `next(s,B)` = $\{\mathtt{t} \mid (\mathtt{s}, \alpha, \mathtt{t}) \in \delta\}$ and `acc(s,B)` = $(\mathtt{s} \in \mathtt{F})$. The two depth-first searches `DFS1` and `DFS2` are interleaved. When `DFS1` is ready to backtrack from an accepting state after completing the search of its successors, it starts `DFS2` in search of a cycle through this state. If `DFS2` fails to find such a cycle, it resumes `DFS1` from the point it was interrupted.

DDFS algorithm
input: Büchi automaton $\mathtt{B} = (\Sigma, \mathtt{Q}, \mathtt{Q}_0, \delta, \mathtt{F})$.
output: **true** if $\mathtt{L(B)} \neq \emptyset$; **false** otherwise.

```
(1)  for all (q0 ∈ init(B)) if (DFS1(q0)) return true;
(2)  return false;
```

DFS1 algorithm
global: B
input: State s ∈ Q.
output: **true** if accepting cycle is reachable from s; **false** otherwise.

```
(1)  add (s,0) to HashTbl;
(2)  add s to Stack;
(3)  for all (t ∈ next(s,B)) if ((t,0) ∉ HashTbl && DFS1(t)) return true;
(4)  if (acc(s,B) && (t,1) ∉ HashTbl && DFS2(s)) return true;
(5)  delete s from Stack;
(6)  return false;
```

DFS2 algorithm
global: B, HashTbl, Stack.
input: State s ∈ Q.
output: **true** if s is in a cycle; **false** otherwise.

```
(1)  add (s,1) to HashTbl;
(2)  for all (t ∈ next(s,B))
(3)    if (t ∈ Stack) return true;
(4)    if ((t,1) ∉ HashTbl && DFS2(t)) return true;
(5)  return false;
```

The state transition graph of a concurrent system S can be represented as a Büchi automaton B_S. Assuming that S is specified succinctly, then $|B_S| = O(2^{|S|})$. One can avoid the explicit construction of B_S by generating the states in $\mathtt{init}(\mathtt{B_S})$ and $\mathtt{next}(\mathtt{s}, \mathtt{B_S})$ on demand and performing the test for acceptance $\mathtt{acc}(\mathtt{s}, \mathtt{B_S})$

symbolically. This *on-the-fly* approach considerably improves the space requirements of DDFS, since it constructs only the reachable part of B_S.

3 Optimal Monte Carlo Estimation

Many engineering and computer science applications require the computation of the mean value μ_Z for a random variable Z distributed in $[0,1]$. When an exact computation of μ_Z proves intractable, being, for example, NP-hard, Monte Carlo methods are often used to compute an (ϵ,δ)-approximation of this quantity. The main idea is to use N independent random variables (or samples) $Z_1,\ldots,Z_N$ identically distributed according to Z with mean μ_Z, and to take $\widetilde{\mu}_Z = (Z_1 + \ldots + Z_N)/N$ as the approximation of μ_Z.

An important issue in such an approximation scheme is determining the value for N. The *zero-one estimator theorem* [17] guarantees that if N is proportional to $\Upsilon = 4\ln(2/\delta)/\mu_Z\epsilon^2$ then $\widetilde{\mu}_Z$ approximates μ_Z with absolute error ϵ and with probability $1-\delta$. More precisely:

$$\mathbf{Pr}[\mu_Z(1-\epsilon) \leq \widetilde{\mu}_Z \leq \mu_Z(1+\epsilon)] \geq 1-\delta$$

When applying the zero-one estimator theorem, one encounters, however, two main difficulties. The first is that N depends on $1/\mu_Z$, the inverse of the value that one intends to approximate. This problem can be circumvented by finding an upper bound κ of $1/\mu_Z$ and using κ to compute N. Finding a tight upper bound is however in most cases very difficult, and a poor choice of κ leads to a prohibitively large value for N. An ingenious way of computing N without relying on μ_Z or κ is provided by the *Stopping Rule Algorithm* (SRA) of [8]. When $\mathbf{E}[Z] = \mu_Z > 0$ and $\Sigma_{i=1} Z_i \geq \Upsilon$, the expectation $\mathbf{E}[N]$ of N equals Υ.

SRA algorithm
input: (ϵ,δ) `with` $0<\epsilon<1$ `and` $\delta>0$.
input: `Random vars` Z_i `with` $i>0$, `independent and identically distributed.`
output: $\widetilde{\mu}_Z$ `approximation of` μ_Z.

```
(1)  Υ = 4 (e - 2) ln(2 / δ) /ε²; Υ1 = 1 + (1 + ε) Υ;
(2)  for (N = 0, S = 0; S ≤ Υ1; N++) S = S + Z_N;
(3)  μ̃_Z = S/N; return μ̃_Z;
```

The second difficulty in applying the zero-one estimator theorem is the factor $1/\mu_Z\epsilon^2$ in the expression for Υ, which can render the value of N unnecessarily large. A more practical approach is offered by the *generalized zero-one estimator theorem* of [8] which states that N is proportional to $\Upsilon' = 4\rho_Z \ln(2/\delta)/(\mu_Z\epsilon)^2$ where $\rho_Z = \max\{\sigma_Z^2, \epsilon\mu_Z\}$ and σ_Z^2 is the variance of Z. Thus, if σ_Z^2, which equals $\mu_Z(1-\mu_Z)$ for Z a Bernoulli random variable, is greater than $\epsilon\mu_Z$, then $\sigma_Z^2 \approx \mu_Z$, $\rho_Z \approx \mu_Z$ and therefore $\Upsilon' \approx \Upsilon$. If, however, σ_Z^2 is smaller than $\epsilon\mu_Z$, then $\rho_Z = \epsilon\mu_Z$ and Υ' is smaller than the Υ by a factor of $1/\epsilon$.

To obtain an appropriate bound in either case, [8] have proposed the *optimal approximation algorithm* (OAA) shown above. This algorithm makes use of the

OAA algorithm
input: Error margin ϵ and confidence ratio δ with $0 < \epsilon \leq 1$ and $0 < \delta \leq 1$.
input: Random vars Z_i, Z'_i with $i > 0$, indep. and identically distrib.
output: $\widetilde{\mu}_Z$ approximation of μ_Z.

(1) $\Upsilon = 4\,(e-2)\ln(2/\delta)/\epsilon^2$; $\Upsilon_2 = 2\,(1+\sqrt{\epsilon})\,(1+2\sqrt{\epsilon})\,(1+\ln(3/2)/\ln(2/\delta))\,\Upsilon$;

(2) $\widehat{\mu}_Z$ = SRA(min{1/2, $\sqrt{\epsilon}$}, δ/3, Z);

(3) $N = \Upsilon_2\,\epsilon/\widehat{\mu}_Z$; $S = 0$;
(4) **for** (i = 1; i $\leq$ N; i++) $S = S + (Z'_{2i-1} - Z'_{2i})^2/2$;
(5) $\widehat{\rho}_Z = \max\{S/N, \epsilon\,\widehat{\mu}_Z\}$;

(6) $N = \Upsilon_2\,\widehat{\rho}_Z/\widehat{\mu}_Z^2$; $S = 0$;
(7) **for** (i = 1; i $\leq$ N; i++) $S = S + Z_i$;

(8) $\widetilde{\mu}_Z = S/N$; **return** $\widetilde{\mu}_Z$;

outcomes of previous experiments to compute N, a technique also known as *sequential analysis*. The OAA algorithm consists of three steps. The first step calls the SRA algorithm with parameters $(\sqrt{\epsilon}, \delta/3)$ to get an estimate $\widehat{\mu}_Z$ of μ_Z. The choice of parameters is based on the assumption that $\rho_Z = \epsilon\mu_Z$, and ensures that SRA takes $3/\epsilon$ less samples than would otherwise be the case. The second step uses $\widehat{\mu}_Z$ to get an estimate of $\widehat{\rho}_Z$. The third step uses $\widehat{\rho}_Z$ to get the desired value $\widetilde{\mu}_Z$. Should the assumption $\rho_Z = \epsilon\mu_Z$ fail to hold, the second and third steps will compensate by taking an appropriate number of additional samples. As shown in [8], OAA runs in an expected number of experiments that is within a constant factor of the minimum expected number.

4 The Quantitative Model-Checking Algorithm

In this section, we present our randomized, automata-theoretic approach to model checking based on the DDFS (Section 2) and Monte Carlo OAA (Section 3) algorithms. The *samples* we are interested in are the reachable cycles (or "lassos") of a Büchi automaton B.[3] Should B be the product automaton $B_S \times B_{\neg\varphi}$ defined in Section 2, then a lasso containing a final state of B (an "accepting lasso") can be interpreted as a *counter-example* to $S \models \varphi$. A lasso of B is sampled via a random walk through B's transition graph, starting from a randomly selected initial state of B.

Definition 5 (Lasso sample space). *A finite run $\sigma = s_0x_0 \ldots s_nx_ns_{n+1}$ of a Büchi automaton $B = (\Sigma, Q, Q_0, \delta, F)$, is called a* lasso *if $s_0 \ldots s_n$ are pairwise distinct and $s_{n+1} = s_i$ for some $0 \leq i \leq n$. Moreover, σ is said to be an* accepting lasso *if some $s_j \in F$, $i \leq j \leq n$; otherwise it is a* non-accepting lasso. *The* lasso sample space *U of B is the set of all lassos of B, while U_a and U_n are the sets of all accepting and non-accepting lassos of B, respectively.*

Definition 6 (Run probability). *The* probability $\mathbf{Pr}[\sigma]$ *of a finite run $\sigma = s_0x_0 \ldots s_{n-1}x_{n-1}s_n$ of a Büchi automaton B is defined inductively as follows:*

[3] We assume without loss of generality that every state of a Büchi automaton B has at least one outgoing transition, even if this transition is a self-loop.

$\mathbf{Pr}[s_0] = k^{-1}$ *if* $|Q_0| = k$ *and* $\mathbf{Pr}[s_0 x_0 \ldots s_{n-1} x_{n-1} s_n] = \mathbf{Pr}[s_0 x_0 \ldots s_{n-1}] \cdot \pi[s_{n-1} x_{n-1} s_n]$ *where* $\pi[s\,x\,t] = m^{-1}$ *if* $(s, x, t) \in \delta$ *and* $|\delta(s)| = m$.

Note that the above definition explores uniformly outgoing transitions. An alternative definition would explore uniformly successor states.

Example 1 (Probability of lassos). Consider the Büchi automaton B of Figure 1. It contains four lassos, `11`, `1244`, `1231` and `12344`, having probabilities 1/2, 1/4,

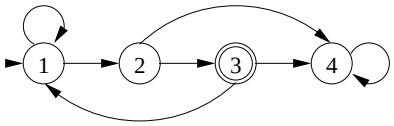

Fig. 1. Example lasso probability space.

1/8 and 1/8, respectively. Lasso `1231` is accepting.

Proposition 1 (Probability space). *Given a Büchi automaton B, the pair $(\mathcal{P}(U), \mathbf{Pr})$ defines a* discrete probability space.

The proof first considers the infinite tree T corresponding to the infinite unfolding of δ. T' is the (finite) tree obtained by making a cut in T at the first repetition of a state along any path in T. It is easy to show by induction on the height of T' that the sum of the probabilities of the runs (lassos) associated with the leaves of T' is 1.

Definition 7 (Random variable). *The* Bernoulli random variable Z *associated with the probability space $(\mathcal{P}(U), \mathbf{Pr})$ of a Büchi automaton B is defined as follows: $p_Z = \mathbf{Pr}[Z{=}1] = \sum_{\lambda_n \in U_n} \mathbf{Pr}[\lambda_n]$ and $q_Z = \mathbf{Pr}[Z{=}0] = \sum_{\lambda_a \in U_a} \mathbf{Pr}[\lambda_a]$ where λ_a is an accepting lasso and λ_n is a non-accepting lasso.*

Example 2 (Bernoulli random variable). For the Büchi automaton B of Figure 1, the lassos Bernoulli variable has associated probabilities $p_Z = 7/8$ and $q_Z = 1/8$.

The *expectation* (or weighted mean) $\mu_Z = 0 \cdot q_Z + 1 \cdot p_Z$ of Z is equal to p_Z. It, or more precisely $1 - p_Z$, provides a measure of the number of counterexamples (accepting lassos) in B, weighted by their probability. Since an exact computation of p_Z is often intractable due to state explosion, we compute an (ϵ,δ)-approximation $\widetilde{p}_Z$ of p_Z using the `OAA` algorithm. We then use $\widetilde{p}_Z$ to derive a Monte Carlo decision procedure we call `QMC` (*Quantitative Model Checking*) for the LTL model-checking problem. `QMC` works as follows: (1) Take independent random samples (lassos) Z_i and Z'_i, each identically distributed according to Z with mean p_Z as required by `OAA`. (2) If an accepting lasso is encountered, break and return the lasso as a counterexample. (3) If all samples are non-accepting, conclude that p_Z is 1 with error margin ϵ and confidence ratio δ. Our use of `OAA` yields a one-sided-error decision procedure for the LTL model-checking problem as `QMC` correctly decides false if $\widetilde{p}_Z < 1$. The `QMC` algorithm is given below, where `rInit(B)=random(`$\mathtt{Q}_0$`)`, `rNext(s,B) =t`$'$ s.t.

MC^2 algorithm
input: Büchi automaton $\mathtt{B} = (\Sigma, \mathtt{Q}, \mathtt{Q}_0, \delta, \mathtt{F})$;
input: Error margin ϵ and confidence ratio δ with $0 < \epsilon \leq 1$ and $0 < \delta \leq 1$.
output: Either (false, lasso l) or (true, $\mathbf{Pr}[1/(1+\epsilon) \leq p_Z] \geq 1-\delta$)

```
(1)  try {p̃_Z = OAA(ε, δ, RACV(B)); return (true, Pr[1/(1+ε) ≤ p_Z] ≥ 1−δ);}
(2)  catch(l) { return (false,l);}
```

RACV algorithm
input: Büchi automaton B;
output: Samples a random cycle of B;
Throws HashTbl if cycle is accepting; returns 1 otherwise.

```
(1)  s := rInit(B); i := 1; f := 0;
(2)  while (s ∉ HashTbl) {
(3)    HashTbl(s) := i;
(4)    if (acc(s,B)) f := i;
(5)    s := rNext(B,s); i := i+1; }
(6)  if (HashTbl(s) ≤ f) throw(lasso(HashTbl)) else return 1;
```

$(\mathtt{s}, \alpha, \mathtt{t}') = \mathtt{random}(\{\tau \in \delta \mid \exists \alpha, \mathtt{t}.\tau = (\mathtt{s}, \alpha, \mathtt{t})\})$, and `acc(s,B)=(s ∈ F)`. The main routine consists of a single statement in which the `OAA` algorithm is called with parameters ϵ, δ, and the *random accepting cycle variable* (`RACV`) routine, which generates on demand the random samples Z_i and Z_i' used in `OAA` as follows. A random lasso is generated using the *randomized init* (`rInit`) and *randomized next* (`rNext`) routines. To determine if the generated lasso is accepting, we store the index `i` of each encountered state `s` in `HashTbl` and record the index of the most recently encountered accepting state in `f`. When we find a cycle, i.e., the state returned by `rNext(M,s)` is in `HashTbl`, we check if `HashTbl(t)` $\leq$ `f`; the cycle is an accepting cycle if and only if this is the case. The function `lasso()` extracts a lasso from the states stored in `HashTbl`.

`QMC` takes as input an explicit representation B of a Büchi automaton. As with `DDFS`, given a succinct representation S of B, one can avoid the explicit construction of B by generating random states `rInit(B)` and `rNext(s,B)` *on the fly* from S, and performing the test for acceptance `acc(s,B)` symbolically. In Section 5 we present such a succinct representation and show how to efficiently generate random initial and successor states.

Theorem 1 (`QMC` correctness). *Given a Büchi automaton B, error margin ϵ, and confidence ratio δ, $\widetilde{p}_Z$, the (ϵ,δ)-approximation of p_Z computed by* `QMC` *is such that if $\widetilde{p}_Z < 1$ then $L(B) \neq \emptyset$, and if $\widetilde{p}_Z = 1$ then $\mathbf{Pr}[1/(1+\epsilon) \leq p_Z] \geq 1-\delta$.*[4]

Proof. The `OAA` algorithm of [8] requires that all samples Z_i, Z_i' are independent of one another and have the same mean value. Independence in our case follows from the fact that each call to `RACV` can be shown to be an independent Bernoulli trial. Moreover, all samples (random lassos) have the same mean value p_Z. Now, if an accepting lasso is found, $L(B) \neq \emptyset$ by definition. Otherwise, $\widetilde{p}_Z = 1$ and

[4] The theorem uses Bayesian logic to express our confidence that, if after N samples `QMC` does not find a counter-example, $\mathbf{Pr}[1/(1+\epsilon) \leq p_Z] \geq 1-\delta$. An alternative formulation uses statistical hypothesis testing.

the result follows from $\mathbf{Pr}[p_Z(1-\epsilon) \leq \widetilde{p}_Z \leq p_Z(1+\epsilon)] \geq 1-\delta$ by observing that $p_Z(1-\epsilon) \leq \widetilde{p}_Z$ is a tautology, and dividing what remains by $1+\epsilon$.

For ϵ sufficiently small and $p_Z = 1 - q_Z$, the above theorem can be rewritten to yield the following upper bound ϵ on the expectation of an accepting lasso in B: $\mathbf{Pr}[q_Z < \epsilon] \geq 1-\delta$. In other words, if `QMC` fails to find a counter-example, the probability of one is bounded from above by ϵ with high probability.

`QMC` is very efficient in both time and space. The *recurrence diameter* of a Büchi automaton B is the longest initialized loop-free path in B.

Theorem 2 (`QMC` complexity). *Let B be a Büchi automaton, D its recurrence diameter and $N = O(4\ln(2/\delta)/\epsilon)$ be the number of samples taken by `OAA` when all Z_i and Z_i' return 1, for a given ϵ, δ. Then, `QMC` takes time $O(N \cdot D)$ and uses space $O(D)$.*

Proof. The length of a lasso is bounded by D; the number of samples taken by `OAA` is bounded by N.

`QMC` can also be run in *estimator mode*, where rather than halting upon finding a counter-example, continues sampling until $\widetilde{p}_Z$ has been computed. By virtue of its reliance on the `OAA` algorithm, `QMC` in estimator mode may not terminate if the number of initialized non-accepting cycles in B is less than Υ_1. Should this not be the case, however, `QMC` provides an estimate of how "false" is the judgement $S \models \varphi$, a useful statistical measure.

5 Implementation

We have implemented the `DDFS` and `QMC` algorithms as an extension to jMocha [1], a model checker for synchronous and asynchronous concurrent systems specified using *reactive modules* [2]. An LTL formula $\neg\varphi$ is specified in our extension of jMocha as a pair consisting of a reactive module monitor and a boolean formula defining its set of accepting states. By selecting the new enumerative or randomized LTL verification option one can check whether $S \models \varphi$: jMocha takes their composition and applies either `DDFS` or `QMC` on-the-fly to check for accepting lassos.

An example reactive module, for a "fair stick" in the dining philosophers problem, is shown below. It consists of a collection of typed variables partitioned into *external* (input), *interface* (output) and *private*. For this example, `rqL`, `rqR`, `rlR`, `rlR`, `grL`, `grR`, `pc`, and `pr` denote left and right request, left and right release, program counter, and priority, respectively. The priority variable `pr` is used to enforce fairness. The values `l`, `r` and `f` stand for left, right and free, respectively.

Variables change their values in a sequence of rounds. The first is an *initialization* round; the subsequent are *update* rounds. Initialization and updates of controlled (interface and private) variables are specified by *actions* defined as a set of *guarded parallel assignments*. Moreover, controlled variables are partitioned into *atoms*: each variable is initialized and updated by exactly one atom.

The initialization round and all update rounds are divided into subrounds, one for the environment and one for each atom A. In an A-subround of the initialization round, all variables controlled by A are initialized simultaneously, as defined by an initial action. In an update A-subround, all variables controlled by A are updated simultaneously, as defined by an update action.

```
type stickType is {f,l,r}
module Stick is
  external rqL,rqR,rlL,rlR:event;
  interface grL,grR:event; private pc,pr:stickType;
  atom STICK
    controls pc,pr,grL,grR
    reads pc,pr,grL,grR,rqL,rqR,rlL,rlR awaits rqL,rqR,rlL,rlR
    init
      [] true -> pc' := f; pr' := l;
    update
      [] pc = f & rqL? & ¬rqR?        -> grL!; pc':= l; pr' := r;
      [] pc = f & rqL? & rqR? & pr = l -> grL!; pc':= l; pr' := r;
      [] pc = f & rqL? & rqR? & pr = r -> grR!; pc':= r; pr' := l;
      [] pc = f & rqR? & ¬rqL?        -> grR!; pc':= r; pr' := l;
      [] pc = l & rlL?                -> pc' := f;
      [] pc = r & rlR?                -> pc' := f;
```

In a round, each variable x has two values: the value at the beginning of the round, written as x and called the *read value*, and the value at the end of the round written as x' and called the *updated value*. *Events* are modeled by toggling boolean variables. For example $\texttt{rqL?} \stackrel{\text{def}}{=} \texttt{rqL}' \neq \texttt{rqL}$ and $\texttt{grL!} \stackrel{\text{def}}{=} \texttt{grL}' := \neg\texttt{grL}$. If a variable x controlled by an atom A depends on the updated value y' of a variable controlled by atom B, then B has to be executed before A. We say that A *awaits* B and that y is an awaited variable of A. The await dependency defines a partial order $\succ$ among atoms.

Operators on modules include *renaming*, *hiding* of output variables, and *parallel composition*. The latter is defined only when the modules update disjoint sets of variables and have a joint acyclic await dependency. In this case, the composition takes the union of the private and interface variables, the union of the external variables (minus the interface variables), the union of the atoms, and the union of the await dependencies.

rNext algorithm
input: Reactive module M; Current state s;
output: Random next state s.all′.

```
(1)  s.extl′ := random(Q.M.extl);
(2)  for all (A ∈ ≻_M^L) {
(3)    for (m := |A.upd|; m ≥ 0; m--) {
(4)      i := random(m);
(5)      if (A.upd(i).grd(s)) break else remove(A.upd,i); }
(6)    if (m = 0) s.ctrl′ := s.ctrl; else s.ctrl′ := random(A.upd(i).ass(s)); }
(7)  return s′;
```

A feature of our QMC implementation is that the next state s′ = rNext(s,M) of M along a random walk in search of an accepting lasso is generated randomly both for the external variables M.extl and for the controlled variables M.ctrl. For the external variables we randomly generate a state s.extl′ in the set of all input valuations Q.M.extl. For the controlled variables we proceed for each atom A in the linear order $\succ_{\mathtt{M}}^{L}$ compatible with $\succ_{\mathtt{M}}$ as follows: first we randomly choose a guarded assignment A.upd(i) with true guard A.upd(i).grd(s), where i is less than the number |A.upd| of guarded assignments in A; then we randomly generate a state s.ctrl′ among the set of all states possibly returned by its parallel (nondeterministic) assignment A.upd(i).ass(s). If no guarded assignment is enabled we keep the current state s.ctrl. The routine rInit is implemented in a similar way.

6 Experimental Results

We compared the performance of QMC and DDFS by applying our implementation of these algorithms in jMocha to the dining philosophers problem. All reported results were obtained on a PC equipped with an Athlon 2100+ MHz processor and 1GB RAM running Linux 2.4.18 (Fedora Core 1).

For dining philosophers, we considered two LTL properties: *deadlock freedom* (DF), which is a safety property, and *starvation freedom* (SF), which is a liveness property. For a system of n philosophers, their specification is as follows:

$$\mathtt{DF} : \mathtt{G} \neg (\mathtt{pc_1} = \mathtt{wait} \,\&\, \ldots \,\&\, \mathtt{pc_n} = \mathtt{wait})$$
$$\mathtt{SF} : \mathtt{G}\,\mathtt{F}\,(\mathtt{pc_1} = \mathtt{eat})$$

We encoded our solution to the problem using Reactive Modules, developing both a symmetric and asymmetric version. In the symmetric case, all philosophers can simultaneously pick up their right forks, leading to deadlock. Lockout-freedom is also violated since no notion of fairness has been incorporated into the solution. That both properties are violated is intentional, as it allow us to compare the relative performance of DDFS and QMC on finding counter-examples. We ran QMC in both decision and estimation modes.

	DDFS		QMC			
ph	time	entr	time	mxl	cxl	N
4	0.02	31	0.08	10	10	3
8	1.62	511	0.20	25	8	7
12	3:13	8191	0.25	37	11	11
16	>20:0:0	–	0.57	55	8	18
20	–	oom	3.16	484	9	20
30	–	oom	35.4	1478	11	100
40	–	oom	11:06	13486	10	209

	DDFS		QMC			
ph	time	entr	time	mxl	cxl	N
4	0.17	29	0.02	8	8	2
8	0.71	77	0.01	7	7	1
12	1:08	125	0.02	9	9	1
16	7:47:0	173	0.11	18	18	1
20	–	oom	0.06	14	14	1
30	–	oom	1.12	223	223	1
40	–	oom	1.23	218	218	1

Table 1. Deadlock and Starvation freedom for symmetric (unfair) version.

For the symmetric case, we chose a value of 10^{-1} for both ϵ and δ, resulting in $N = 1257$ samples taken. This number of samples proved sufficiently large in that for each instance of dining philosophers on which we ran our implementation of `QMC`, a counter-example was detected. The results are given in Table 1. The meaning of the column headings is the following: `ph` is the number of philosophers; `time` is the time to find a counter-example in `hrs:mins:secs`; `entr` is the number of entries in the hash table; `mxl` is the maximum length of a sample; `cxl` is the the length of the counter-example; `N` is the number of samples taken.

As the data in the tables demonstrate, `DDFS` runs out of memory for 20 philosophers, while `QMC` not only scales up to a larger number of philosophers, but also outperforms `DDFS` on the smaller numbers. This is especially the case for starvation freedom where one sample is enough to find a counter-example.

One might wonder why `DDFS` spends more than 7 hours to check for starvation freedom on 16 philosophers while the number of entries in the hash table, which can be understood as the stack depth in the depth-first search, is only 173? Or why does it run out of memory for 20 or more philosophers? The reason is that `init(B)`, which is called by `DDFS`, and `next(B,s)`, which is called at each recursive invocation of `DDFS1` and `DDFS2`, may generate a large number of successor states. As a consequence, each path stored in the hash table may have associated with it a number of states stored in temporary variables that is considerably larger than the path length.

To avoid storing a large number of states in temporary variables, one might attempt to generate successor states one at a time (which exactly what `rNext(B,s)` of `QMC` does). However, the constraint imposed by `DDFS` to generate *all* successor states in sequential order inevitably leads to the additional time and memory consumption.

phi.	N	satisf.	avg.len.	counter.	avg.len.	$\widetilde{p}_Z$
4	6169	3957	7.827	2212	7.271	0.637
5	5526	3834	9.122	1692	7.779	0.688
6	4922	3590	10.225	1332	8.113	0.731
7	3975	3048	11.446	927	8.311	0.759
8	3593	2911	12.187	682	8.472	0.808
9	2959	2481	13.363	478	8.964	0.836
10	2884	2454	14.531	430	8.874	0.855

Table 2. Variation of $\widetilde{p}_Z$ for DF with respect to the number of philosophers.

In estimation mode, we set $\epsilon = 10^{-1}$, $\delta = 10^{-2}$ and checked for deadlock freedom on all samples, as required by `OAA`, without returning at the first counter-example. This allowed us to compute values for $\widetilde{p}_Z$ for varying numbers of philosophers; our results are given in Table 1. Observe that $\widetilde{p}_Z$, which can be interpreted as an estimate of the probability that an arbitrary run of a symmetric system of n dining philosophers is deadlock-free, increases apparently linearly with n. This observation is fairly obvious in retrospect, but to our knowledge has not been reported previously in the literature.

	DDFS		QMC		
phi	time	entries	time	mxl	avl
4	0:01	178	0:20	49	21
6	0:03	1772	0:45	116	42
8	0:58	18244	2:42	365	99
10	16:44	192476	7:20	720	234
12	–	oom	21:20	1665	564
14	–	oom	1:09:52	2994	1442
16	–	oom	3:03:40	7358	3144
18	–	oom	6:41:30	13426	5896
20	–	oom	19:02:00	34158	14923

	DDFS		QMC		
phi	time	entries	time	mxl	avl
4	0:01	538	0:20	50	21
6	0:17	9106	0:46	123	42
8	7:56	161764	2:17	276	97
10	–	oom	7:37	760	240
12	–	oom	21:34	1682	570
14	–	oom	1:09:45	3001	1363
16	–	oom	2:50:50	6124	2983
18	–	oom	8:24:10	17962	7390
20	–	oom	22:59:10	44559	17949

Table 3. Deadlock and starvation freedom for the fair asymmetric version.

In the asymmetric case, a notion of fairness has been incorporated into the specification and, as a result, deadlock and starvation freedom are preserved. Specifically, the specification uses a form of round-robin scheduling to explicitly encode weak fairness. As in the symmetric case, we chose the value 10^{-1} for both ϵ and δ. Our results are given in Table 3, where columns `mxl` and `avl` represent the maximum and the average length of a sample, respectively.

Dining philosophers is a well known benchmark and its state space can be easily manipulated. Systems such as these have been shown to be amenable to verification techniques such as abstraction and symmetry reduction. Nevertheless, abstraction approaches often require human intervention, while symmetry reduction requires an underlying symmetry to be present in the system structure. In any event, techniques such as abstraction and symmetry reduction are orthogonal concepts to Monte Carlo model checking; the `QMC` algorithm could take advantage of them, as well.

7 Related Work

There have been a number of prior proposals for randomized approaches to the model-checking problem. Like our `QMC` algorithm, the Lurch debugger [21, 14] performs random sampling in search of initialized random cycles; it also searchs for initialized random terminal paths. Lurch does not, however, compute an (ϵ, δ)-approximation like `QMC` does. Rather it randomly searches the system's state space until a "saturation point" or a user-defined limit on time or memory is reached. Moreover, it appears that the system is only checking safety properties; `QMC`, on the other hand, is a Monte Carlo model checker for general LTL formulas.

In [4] randomization is used to decide which visited states should be stored, and which should be omitted, during LTL model checking, with the goal of reducing memory requirements.

Probabilistic model checkers cater to stochastic models and logics, including, but not limited to, those for discrete- and continuous-time Markov chains [18, 3], Probabilistic I/O Automata [26], and Probabilistic Automata [23]. Like `QMC`, these model checkers return results of a statistical nature.

Stochastic modeling formalisms and logics are also considered in [29, 15, 24], who advocate an approach to the model checking based on random sampling of execution paths and statistical hypothesis testing. In particular, [15] uses bounded model checking to bound the length of sampled execution paths in the course of computing an (ϵ, δ)-approximation for the "positive LTL" fragment of LTL. The number of samples taken is $4\log(2/\delta)/\epsilon^2)$. In contrast, our `QMC` algorithm is applicable to the classical model-checking problem for nondeterministic/concurrent systems and general LTL formulas, performs random sampling of lassos, and uses a number of samples that is optimal to within a constant factor.

Several techniques have been proposed for the automatic verification of safety and reachability properties of concurrent systems based on the use of random walks to uniformly sample the system state space [19, 13, 27]. In contrast, `QMC` performs random sampling of lassos for general LTL model checking. In [20], Monte Carlo and abstract interpretation techniques are used to yield upper bounds on the probability of certain outcomes of programs whose inputs are probabilistic or nondeterministic.

8 Conclusions

We have presented `QMC`, a randomized, Monte Carlo decision procedure for classical temporal-logic model checking. Utilizing the optimal algorithm of [8] for Monte Carlo estimation, `QMC` performs random sampling of lassos in the Büchi automaton $B = B_S \times B_{\neg\varphi}$ to yield a one-sided error Monte Carlo decision procedure for the LTL model-checking problem $S \models \varphi$. It does so using a number of samples N that is optimal to within a constant factor. It also delivers quantitative information about the model-checking problem in the form of an (ϵ, δ)-approximation of the expectation that $L(B) = \emptyset$. Benchmarks show that `QMC` is fast, memory-efficient, and scales extremely well.

To take a random sample, which in our case is a random lasso, `QMC` performs a "uniform" random walk through B: one in which the next transition taken is decided by tossing a fair, k-sided coin when a state of B is reached having k outgoing transitions. This can lead to assigning lassos probabilities that may not reflect actual system behavior. This potential problem is mitigated if the state-transition behavior of a system S is prescribed by a probabilistic automaton such as a discrete Markov chain M, as in probabilistic model checking. In this case, there is a natural way to assign a probability to a random walk σ: it is simply the product of the state-transition probabilities p_{ij} for each transition from state i to j along σ. This implies that `QMC` extends with little modification to the case of probabilistic model checking.

Another way to obtain a Monte Carlo decision procedure for LTL model checking is to appeal directly to the theory of geometric random variables to determine the number of samples needed to find an accepting lasso with probability at least $1 - \delta$. This is the approach taken in [12], the advantage of which is that it usually takes significantly fewer samples than that required by `OAA`. On the other hand, it forgoes the computation of an (ϵ, δ)-approximation of p_Z.

References

1. R. Alur, L. de Alfaro, R. Grosu, T. A. Henzinger, M. Kang, C. M. Kirsch, R. Majumdar, F. Mang, and B. Y. Wang. JMOCHA: A model checking tool that exploits design structure. In *Proceedings of the 23rd international conference on Software engineering*, pages 835–836. IEEE Computer Society, 2001.
2. R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, July 1999.
3. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes. In *Proc. of TACAS*, 2004.
4. L. Brim, I. Černá, and M. Nečesal. Randomization helps in LTL model checking. In *Proceedings of the Joint International Workshop, PAPM-PROBMIV 2001*, pages 105–119. Springer, LNCS 2165, September 2001.
5. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
6. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, LNCS 131, pages 52–71. Springer, 1981.
7. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2-3):275–288, 1992.
8. P. Dagum, R. Karp, M. Luby, and S. Ross. An optimal algorithm for Monte Carlo estimation. *SIAM Journal on Computing*, 29(5):1484–1496, 2000.
9. P. Dagum and M. Luby. An optimal approximation algorithm for bayesian inference. *Artificial Intelligence*, 78:1–27, 1997.
10. M. Dyer, A. Frieze, and R. Kannan. A random polynomial time algorithm for approximating the volume of convex bodies. In *Proceedings of the 21st IEEE Symposium on the Theory of Computing*, pages 375–381, 1989.
11. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
12. R. Grosu and S. A. Smolka. Monte Carlo model checking. Technical report, Department of Computer Science, SUNY Stony Brook, 2004. `www.cs.sunysb.edu/∼sas/papers/GS04.pdf`.
13. P. Haslum. Model checking by random walk. In *Proc. of 1999 ECSEL Workshop*, 1999.
14. M. Heimdahl, J. Gao, D. Owen, and T. Menzies. On the advantages of approximate vs. complete verification: Bigger models, faster, less memory, usually accurate. In *Proc. of 28th Annual NASA Goddard Software Engineering Workshop (SEW'03)*, 2003.
15. T Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *Proc. Fifth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004)*, 2004.
16. G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. of the Second Spin Workshop*, pages 23–32, 1996.
17. R. Karp, M. Luby, and N. Madras. Monte-Carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10:429–448, 1989.
18. M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In *Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*, pages 200–204. Springer-Verlag, 2002.

19. M. Mihail and C. H. Papadimitriou. On the random walk method for protocol testing. In *6th International Conference on Computer Aided Verification (CAV)*, pages 132–141. Springer, LNCS 818, 1994.
20. D. Monniaux. An abstract monte-carlo method for the analysis of probabilistic programs. In *Proc. 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–101. ACM Press, 2001.
21. D. Owen, T. Menzies, M. Heimdahl, and J. Gao. Finding faults quickly in formal models using random search. In *Proc. of SEKE 2003*, 2003.
22. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag.
23. R. Segala and N. A. Lynch. Probabilistic simulations for probabilistic processes. In B. Jonsson and J. Parrow, editors, *Proceedings of CONCUR '94 — Fifth International Conference on Concurrency Theory*, pages 481–496. Volume 836 of *Lecture Notes in Computer Science*, Springer-Verlag, 1994.
24. K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *16th International Conference on Computer Aided Verification (CAV 2004)*, 2004.
25. A. P. Sistla and E. A. Emerson. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32:733–749, 1985.
26. E. W. Stark and S. A. Smolka. Compositional analysis of expected delays in networks of probabilistic I/O automata. In *Proc. 13th Annual Symposium on Logic in Computer Science*, pages 466–477, Indianapolis, IN, June 1998. IEEE Computer Society Press.
27. E. Tronci, G., D. Penna, B. Intrigila, and M. Venturini. A probabilistic approach to automatic verification of concurrent systems. In *Proc. of 8th IEEE Asia-Pacific Software Engineering Conference (APSEC)*, 2001.
28. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.
29. H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *Proc. 14th International Conference on Computer Aided Verification*, 2002.