Monte Carlo Model Checking

Radu Grosu and Scott A. Smolka

Dept. of Computer Science, Stony Brook Univ., Stony Brook, NY, 11794, USA
E-mail: {grosu,sas}@cs.sunysb.edu
Tel: (631) 632-8453 Fax: (631) 632-8334

Abstract. We present $\texttt{MC}^2$, what we believe to be the first randomized, Monte Carlo algorithm for temporal-logic model checking, the classical problem of deciding whether or not a property specified in temporal logic holds of a system specification. Given a specification S of a finite-state system, an LTL (Linear Temporal Logic) formula φ, and parameters ϵ and δ, $\texttt{MC}^2$ takes $N = \ln(\delta)/\ln(1-\epsilon)$ random samples (random walks ending in a cycle, i.e *lassos*) from the Büchi automaton $B = B_S \times B_{\neg\varphi}$ to decide if $L(B) = \emptyset$. Should a sample reveal an accepting lasso l, $\texttt{MC}^2$ returns false with l as a witness. Otherwise, it returns true and reports that with probability less than δ, $p_Z < \epsilon$, where p_Z is the expectation of an accepting lasso in B. It does so in time $O(N \cdot D)$ and space $O(D)$, where D is B's recurrence diameter, using a number of samples N that is optimal to within a constant factor. Our experimental results demonstrate that $\texttt{MC}^2$ is fast, memory-efficient, and scales very well.

Keywords: Model Checking, Monte Carlo

1 Introduction

Model checking [38, 9], 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. A comprehensive discourse on model checking can be found in [11].

Model checking, however, is not without its drawbacks, the most prominent of which is *state explosion*: the phenomenon where the size of a system's state space grows exponentially in the size of its specification. That is, given a succinct representation S of a concurrent system in a modeling formalism such as process algebra [5], Reactive Modules [3], or I/O automata [32], the size of the system's state transition graph (or Kripke structure) K is such that $|K| = O(2^{|S|})$. State explosion can render the model-checking problem intractable for many applications of practical interest. See, for example, [**?**], where it is shown that the problem is PSPACE-complete for LTL (Linear Temporal Logic), and [2], who prove a similar result for Reactive Modules.

Over the past two decades, researchers have developed a plethora of techniques (heuristics) aimed at curtailing state explosion, including: *symbolic model checking* [33, 8], which uses Ordered Binary Decision Diagrams (OBDDs) to succinctly represent a system's state space; *partial-order reduction methods* [19, 44, 17], which are based on the observation that executing two independent events in either order results in the same global state; *symmetry reduction* [16, 10, 21], which utilizes the existence of a nontrivial permutation group that preserves the state transition graph to reduce the state space; and *bounded model checking* [6], which searches for a counter-example in executions whose length is bounded by some integer k, and if no bug is found, increases k until either a bug is found, the problem becomes intractable, or some pre-known upper bound is reached.

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 [14], computing the volume of convex bodies [15], solving the Isling model of statistical mechanics [26], evaluating reliability in planar multi-terminal networks [27], and approximating the number of solutions of a DNF formula [28].

Our approach makes use of the following idea from the automata-theoretic technique of Vardi and Wolper [45] 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 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, where a *lasso* is a cycle reachable from an initial state of B, means that S is *not* a model of φ. Moreover, such an accepting lasso can be viewed as a *counter-example* to $S \models \varphi$.

To decide if $L(B)$ is empty, we have developed the $\texttt{MC}^2$ Monte Carlo model-checking algorithm. Underlying the execution of $\texttt{MC}^2$ 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 probability that an arbitrary run of S is an accepting lasso in B. $\texttt{MC}^2$ takes $N = \ln(\delta)/\ln(1-\epsilon)$ random samples Z_i from B, where a random sample can now be understood as an initialized random walk in B terminating in a cycle, i.e., a lasso. Such a random walk is constructed *on-the-fly* in order to avoid the *a' priori* construction of B, which would immediately lead to state explosion. Should a sample Z_i correspond to an accepting lasso l, $\texttt{MC}^2$ returns false with l as a witness. Otherwise, it returns true and reports that with probability less than δ, $p_Z < \epsilon$.

The main features of our $\texttt{MC}^2$ algorithm are the following.

- To the best of our knowledge, $\texttt{MC}^2$ is the first randomized, Monte Carlo algorithm to be proposed in the literature for the classical problem of temporal-logic model checking.
- $\texttt{MC}^2$ 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] $\texttt{MC}^2$ also delivers *quantitative* information about the model-checking problem. Should the random sampling performed by $\texttt{MC}^2$ not reveal an accepting lasso in $B = B_S \times B_{\neg\varphi}$, $\texttt{MC}^2$ returns true and reports that with probability less than δ, $p_Z < \epsilon$. Recall that p_Z is the expectation of an accepting cycle in B, while ϵ and δ are parameters of the algorithm.
- $\texttt{MC}^2$ 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, the number of samples $N = \ln(\delta)/\ln(1-\epsilon)$ taken by $\texttt{MC}^2$ is optimal to within a constant factor.
- Although we present $\texttt{MC}^2$ in the context of the classical model-checking problem for nondeterministic/concurrent systems, the algorithm works with virtually no modification on systems specified using stochastic modeling formalisms such as discrete-time Markov chains.
- We have implemented $\texttt{MC}^2$ 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 a reachable accepting cycle 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 that $\texttt{MC}^2$ 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 considers the requisite probability theory of geometric random variables. Section 4 presents $\texttt{MC}^2$, our Monte Carlo model-checking algorithm. Section 5 describes our jMocha implementation of $\texttt{MC}^2$. Section 6 summarizes our experimental results. Section 7 considers alternative random-sampling strategies to the one currently used by $\texttt{MC}^2$. Section 8 discusses previous approaches to randomized model checking. Section 9 contains our conclusions and directions for future work.

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 [45]. 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]

[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 8, also produce quantitative results.

[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 $L(B_S \times B_{\neg\varphi}) = \emptyset$

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

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

1. *Every $p \in AP$ is a well-formed (wf) LTL formula.*
2. *If φ and ψ are wf LTL formulas, 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 $\mathcal{P}(AP)$. The satisfaction relation $\xi \models \varphi$ is defined 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 *tableau construction* of [18]. 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. $Q^0 = Q_1^0 \times Q_2^0 \times \{0\}$, $\quad Q = Q_1 \times Q_2 \times \{0,1,2\}$, $\quad F = Q_1 \times Q_2 \times \{2\}$,

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. As explained above, the product operator comes into play in the construction of the product automaton $B_S \times B_{\neg\varphi}$. In the simplest case, all states of B_S are accepting and a simple cross-product construction will suffice. In general, however, not all states of B_S are accepting, in particular, when some sort of fairness constraints have been imposed on S. In this case, the more complicated product construction of Definition 4 is required.

DDFS algorithm
input: Büchi automaton $B = (\Sigma, Q, Q_0, \delta, F)$.
output: **true** if $L(B) \neq \emptyset$; **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;
```

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* (accepting lasso). Looking for

such a lasso is usually done by using the *double depth-first search* algorithm `DDFS` [12, 25] shown above, where $\texttt{init(B)} = \texttt{Q}_0$, $\texttt{next(s,B)} = \{\texttt{t} \mid (\texttt{s},\alpha,\texttt{t}) \in \delta\}$ and $\texttt{acc(s,B)} = (\texttt{s} \in \texttt{F})$.

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.

One can avoid the explicit construction of B_S by generating the states in $\texttt{init}(\texttt{B}_\texttt{S})$ and $\texttt{next}(\texttt{s}, \texttt{B}_\texttt{S})$ on demand and performing the test for acceptance $\texttt{acc}(\texttt{s}, \texttt{B}_\texttt{S})$ symbolically. This *on-the-fly* approach may considerably improve the space requirements of `DDFS`, since it constructs only the reachable part of B_S.

3 Monte Carlo Estimation

As discussed in the Introduction, Monte Carlo methods are often used in engineering and computer-science applications to compute an approximation of a solution whose exact computation proves intractable. Given that the LTL model-checking problem $S \models \varphi$, where S is a succinct representation of a concurrent system and φ is an LTL formula, is PSPACE-complete, little effort has been devoted so far to the use of Monte Carlo approximation techniques. While sacrificing a YES/NO answer to $S \models \varphi$, the Monte Carlo approach holds the promise of being extremely efficient in terms of time/memory usage.

As we will show in Section 4, to each instance $S \models \varphi$ of the LTL model-checking problem, one may associate 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 probability that an arbitrary run of S is a counter-example to φ. Since p_Z is hard to compute, one can use Monte Carlo techniques to derive a one-sided error randomized algorithm for LTL model checking.

Given a Bernoulli random variable Z, define the *geometric* random variable X with parameter p_Z whose value is the number of independent trials required until success, i.e., until $Z = 1$. The *probability mass function* of X is $p(N) = \mathbf{Pr}[X = N] = q_Z^{N-1} p_Z$ and the *cumulative distribution function* (CDF) of X is

$$F(N) = \mathbf{Pr}[X \leq N] = \sum_{n \leq N} p(n) = 1 - q_Z^N$$

Requiring that $F(N) \geq 1 - \delta$ yields:

$$N \geq \ln(\delta) / \ln(1 - p_Z)$$

This provides a lower bound on the number of attempts N needed to achieve success (find a counter-example) with confidence ratio δ.

In our case, however, p_Z is in general unknown. Therefore, in addition to requiring that $F(N) \geq 1 - \delta$, we require that $p_Z \geq \epsilon$, which yields:

$$N' \geq \ln(\delta) / \ln(1 - \epsilon) \geq \ln(\delta) / \ln(1 - p_Z) \qquad (*)$$

This gives us, with confidence ratio δ, a lower bound on the number of attempts N' needed to achieve success, with a lower bound of ϵ for p_Z.

An alternative approach to the one just given is to compute an (ϵ, δ)-approximation $\widetilde{p}_Z$ of p_Z; i.e., $\widetilde{p}_Z$ is such that:

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

This can be done using a number of samples N that is optimal to within a constant factor by appealing to the optimal approximation algorithm (`OAA`) of [13]. This is the kind of algorithm we present in [20], which can be run in two modes: "decision mode", which stops at the first counter-example; and "estimation mode", which runs to completion to compute $\widetilde{p}_Z$.

The approach taken here, in contrast, appeals to basic probability theory of Bernoulli and geometric random variables to derive a decision procedure for the LTL model-checking problem. It is more direct than the one based on `OAA` and as a result yields a value of N that is usually much smaller (an order of magnitude) than that required by `OAA`. This is to be expected as the theory underlying `OAA` is based on the more general Chernoff bounds, which are applicable to any random variable encoding a Poisson trial.

4 Monte Carlo Model-Checking Algorithm

In this section, we present our randomized, automata-theoretic approach to model checking based on the `DDFS` algorithm of Section 2 and the basic probability theory of geometric random variables presented in Section 3. 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 inside the cycle (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 Given a Büchi automaton* $B = (\Sigma, Q, Q_0, \delta, F)$, *a finite run* $\sigma = s_0x_0 \ldots s_nx_ns_{n+1}$ *of* B *is called a* lasso *if* $s_0 \ldots s_n$ *are pairwise distinct and* $s_{n+1} = s_i$ *for some* $0 \leq i \leq n$. *Also,* σ *is said to be an* accepting lasso *if some* $s_i \in F$, $0 \leq i \leq n$; *otherwise it is a* non-accepting lasso. *The* lasso sample space L *of* B *is the set of all lassos of* B, *while* L_a *and* L_n *are the sets of all accepting and non-accepting lassos of* B, *respectively.*

To define a probability space over L we show how to compute the probability of a lasso.

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:* $\mathbf{Pr}[s_0] = k^{-1}$ *if* $|Q_0| = k$ *and* $\mathbf{Pr}[s_0x_0 \ldots s_{n-1}x_{n-1}s_n] = \mathbf{Pr}[s_0x_0 \ldots s_{n-1}] \cdot \pi[s_{n-1}x_{n-1}s_n]$ *where* $\pi[s\,x\,t] = m^{-1}$ *if* $(s, x, t) \in \delta$ *and* $|\delta(s)| = m$.

[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.

Note that the above definition explores uniformly outgoing transitions. An alternative definition might 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,

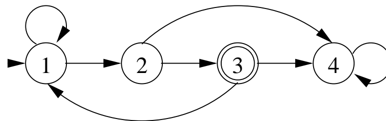

Fig. 1. Example lasso probability space.

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

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

The proof of this proposition 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. *Lasso Bernoulli variable The* random variable Z *associated with the probability space* $(\mathcal{P}(L), \mathbf{Pr})$ *of a Büchi automaton B is defined as follows:* $p_Z = \mathbf{Pr}[Z = 1] = \sum_{\lambda_a \in L_a} \mathbf{Pr}[\lambda_a]$ *and* $q_Z = \mathbf{Pr}[Z = 0] = \sum_{\lambda_n \in L_n} \mathbf{Pr}[\lambda_n]$.

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

Having defined Z, we now present our Monte Carlo decision procedure, which we call `MC`^{2}, for the LTL model-checking problem. Its pseudo-code is as follows, where `acc(s,B)=(s`$\in$`F)`, `rInit(B)=random(`S_0`)`, `rNext(s,B,T)=t`$'$ and `(s,`α'`,t`$'$`)=random({`$\tau \in \delta$ `|` $\exists\, \alpha$`,t.`τ`=(s,`α`,t)})`. The main routine consists of a three statements, the first of which uses inequation (*) of Section 3 to determine the value for N, given parameters ϵ and δ. The second statement is a for-loop that successively samples up to N lassos by calling the *random lasso* (`RL`) routine. If an accepting lasso l is found, `MC`^{2} decides false and returns l as a counter-example. If no accepting lasso is found within N trials, `MC`^{2} decides true, and reports that with probability less than δ, p_Z is less than ϵ.

The `RL` routine generates a random lasso by using the *randomized init* (`rInit`) and *randomized next* (`rNext`) routines. To determine if the generated lasso is accepting, it stores the index `i` of each encountered state `s` in `HashTbl` and records the index of the most recently encountered accepting state in `f`. Upon detecting a cycle, i.e., the state `s:=rNext(B,s)` is in `HashTbl`, it checks if `HashTbl(s)`$\leq$`f`; the cycle is an accepting cycle if and only if this is the case.

MC^2 algorithm
input: $\mathtt{B} = (\Sigma, \mathtt{Q}, \mathtt{Q}_0, \delta, \mathtt{F})$; $0 < \epsilon < 1$; $0 < \delta < 1$.
output: Either (false, counter-example) or (true, "$\mathbf{Pr}[p_Z < \epsilon] \leq \delta$")

```
(1)  N := ln δ / ln (1-ϵ);
(2)  for (i := 1; i ≤ N; i++) if (RL(B)==(1,e)) return (false,e);
(3)  return (true,"Pr[p_Z < ϵ] ≤ δ");
```

RL algorithm
input: Büchi automaton B;
output: Samples a RL. Returns (1, HashTbl) if acc; (0,Null) otherwise

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

As with `DDFS`, one can avoid the explicit construction of B, by generating random states `rInit(B)` and `rNext(B,s)` on demand and performing the test for acceptance `acc(B,s)` symbolically. In the next section we present such a succinct representation and show how to efficiently generate random initial and successor states.

Theorem 1. MC^2 *correctness Given a Büchi automaton B and parameters ϵ and δ, if* MC^2 *returns false, then $L(B) \neq \emptyset$. Otherwise, with probability less than δ, $p_Z < \epsilon$.*

Proof. If RL finds an accepting lasso then $L(B) \neq \emptyset$ by definition. Otherwise, each call to RL can be shown to be an independent Bernoulli trial, and the result follows from inequation (*) of Section 3.

MC^2 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. MC^2 *complexity Let B be a Büchi automaton, D its recurrence diameter and $N = \ln(\delta)/\ln(1-\epsilon)$. Then* MC^2 *runs in time $O(N \cdot D)$ and uses $O(D)$ space. Moreover, N is optimal to within a constant factor.*

Proof. The length of a lasso is bounded by D; the number of samples taken is bounded by N. That N is optimal to within a constant factor follows from inequation (*), which provides a tight lower bound on the number of trials needed to achieve success with confidence ratio δ and lower bound ϵ on p_Z.

It follows from Theorems 1 and 2 that MC^2 is a one-sided error, Monte Carlo decision procedure for the emptiness-checking problem for Büchi automata. For $B = B_S \times B_{\neg\varphi}$, MC^2 yields a Monte Carlo decision procedure for the LTL model-checking problem $S \models \varphi$ requiring $O(N \cdot D)$ time and $O(D)$ space. Compare this with DDFS which runs in $O(2^{|s|+|\varphi|})$ time and space for this problem.

5 Implementation

We have implemented the DDFS and MC^2 algorithms as an extension to jMocha [1], a model checker for synchronous and asynchronous concurrent systems specified using *reactive modules* [3]. 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 MC^2 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.

```
type stickType is {free,left,right}
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' := free; pr' := left;
update
[] pc = free & rqL? & ¬rqR? -> grL!; pc':= left; pr' := right;
[] pc = free & rqL? & rqR? & pr = left -> grL!; pc':= left; pr' := right;
[] pc = free & rqL? & rqR? & pr = right -> grR!; pc':= right; pr' := left;
[] pc = free & rqR? & ¬rqL? -> grR!; pc':= right; pr' := left;
[] pc = left & rlL? -> pc' := free;
[] pc = right & rlR? -> pc' := free;
```

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

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 `rqL?` $\stackrel{\text{def}}{=}$ `rqL'` $\neq$ `rqL` and `grL!` $\stackrel{\text{def}}{=}$ `grL' :=` $\neg$`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 ∈ ≻^L_M) {
(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 $\mathtt{MC}^2$ implementation in jMocha 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 $\succ^L_{\mathtt{M}}$ 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 $\mathtt{MC}^2$ and `DDFS` by applying our implementation of these algorithms in jMocha to the Reactive-Modules specification of two well known model-checking benchmarks: the *dining philosophers* problem and the *Needham Schroeder* mutual authentication protocol. 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:

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

We considered Reactive-Modules specifications of both a symmetric and asymmetric solution to the problem. 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 fariness 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 $\texttt{MC}^2$ on finding counter-examples.

For the symmetric case, we ran $\texttt{MC}^2$ for exactly $N = 1257$ samples, a number arrived at by choosing a value of 10^{-1} for the confidence ratio δ, which, by solving inequation (*) of Section 3 for ϵ, gives a value of $1.8\text{E}-4$ for ϵ, the lower bound on p_Z. This number of samples proved to be sufficiently large in the sense that for each instance of the dining-philosophers problem on which we ran our implementation of $\texttt{MC}^2$, a counter-example was detected. We also computed a lower bound $\widetilde{p}_Z$ on p_Z by solving inequation (*) for ϵ with N' set to the number of samples taken before for the first counter-example was found, and δ set to 10^{-1} as previously mentioned. This yields:

$$\widetilde{p}_Z \geq 1 - e^{ln(\delta)/N}$$

The results for the symmetric unfair case 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-ex. 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 no. samples to find a counter-ex; $\widetilde{\texttt{p}}_{\texttt{Z}}$ is the lower bound computed for p_Z.

	DDFS		$\texttt{MC}^2$				
ph	time	entr	time	mxl	cxl	N	$\widetilde{p}_Z$
4	0.02	31	0.08	10	10	3	0.78
8	1.62	511	0.20	25	8	7	0.48
12	3:13	8191	0.25	37	11	11	0.34
16	>20:0:0	–	0.57	55	8	18	0.22
20	–	oom	3.16	484	9	20	0.20
30	–	oom	35.4	1478	11	100	0.04
40	–	oom	11:06	13486	10	209	0.02

	DDFS		$\texttt{MC}^2$				
ph	time	entr	time	mxl	cxl	N	$\widetilde{p}_Z$
4	0.17	29	0.02	8	8	2	0.90
8	0.71	77	0.01	7	7	1	0.99
12	1:08	125	0.02	9	9	1	0.99
16	7:47:0	173	0.11	18	18	1	0.99
20	–	oom	0.06	14	14	1	0.99
30	–	oom	1.12	223	223	1	0.99
40	–	oom	1.23	218	218	1	0.99

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

As the data in the tables demonstrate, `DDFS` runs out of memory for 20 philosophers, while $\texttt{MC}^2$ 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. As a concrete example, an initial state in the case of 16 philosophers, may have more than 40,000 successors.

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 `MC`^{2} does). However, the constraint imposed by `DDFS` to generate *all* successor states in sequential order inevitably leads to the additional time and memory consumption.

	DDFS		MC^2		
ph	time	entr	time	max	avg
4	0:01	178	0:20	49	21
5	0:01	500	0:27	77	28
6	0:03	1772	0:45	116	42
7	0:11	5344	1:23	188	66
8	0:58	18244	2:42	365	99
9	3:54	57334	4:30	527	151
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		MC^2		
ph	time	entr	time	max	avg
4	0:01	538	0:20	50	21
5	0:03	1986	0:28	79	30
6	0:17	9106	0:46	123	42
7	1:24	36031	1:19	182	64
8	7:56	161764	2:17	276	97
9	43:39	667221	4:47	474	155
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 2. Deadlock and starvation freedom for fair assymetric 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 ran `MC`^{2} for exactly $N = 1257$ samples. Our results are given in Table 2, where columns `mxl` and `avl` represent the maximum and the average length of a sample, respectively.

The next model-checking benchmark we considered was the the Needham-Schroeder public-key authentication protocol; first published in 1978 [36], this protocol initiated a large body of work on the design and analysis of cryptographic protocols. In 1995, Lowe published an attack on the protocol that had apparently been undiscovered for the previous 17 years [30]. The following year, he showed how the flaw could be discovered mechanically by model checking [31].

The intent of the Needham-Schroeder protocol is to establish mutual authentication between principals A and B in the presence of an intruder who can intercept, delay, read, copy, and generate messages, but who does not know the private keys of the principals. The flaw discovered by Lowe uses an interleaving of two runs of the protocol.

To illustrate `MC`^{2}'s ability to find attacks in security protocols like Needham-Schroeder when traditional model checkers fail due to state explosion, we encoded the original (incorrect) Needham-Schroeder protocol as a Reactive-Modules specification and checked if it is free from intruder attacks. Our results are shown

	DDFS		MC^2				
mr	time	entr	time	mxl	cxl	N	$\widetilde{p}_Z$
4	0.38	607	1.68	87	87	103	6.4E-3
8	1.24	2527	11.3	208	65	697	0.9E-3
16	5.87	13471	10.2	223	61	612	1.1E-3
24	18.7	39007	3:06	280	44	12370	5.5E-4
32	36.2	85279	2:54	269	63	11012	6.2E-4
40	1:11	158431	1:46	325	117	7818	8.8E-4
48	2:03	264607	1:45	232	25	6997	9.8E-4
56	3:24	409951	6:54	278	133	28644	2.4E-4
64	5:18	600607	7:12	347	32	29982	2.3E-4
72	–	oom	11:53	336	63	43192	1.6E-4

Table 3. Needham-Schroeder protocol.

in Table 3 where column `mr` represents the the maximum nonce range;[4] i.e., a value of n for `mr` means that a nonce used by the principals can range in value from 0 to n, and also corresponds to the maximum number of runs of the protocol. The meaning of the other columns are the same as those in Tables **??** and 1 for the symmetric (incorrect) version of dining philosophers. The values for $\widetilde{p}_z$ were computed by taking $\delta = 10^{-3}$.

In the case of Needham-Schroeder, counter-examples have a lower probability of occurrence and `DDFS` outperforms `MC`^{2} when the range of nonces is relatively small. However, `MC`^{2} scales up to a larger number of nonces whereas `DDFS` runs out of memory.

We have chosen the above benchmarks for practical purposes: they are well-known, and more importantly, they allowed us to easily manipulate (increase) the state-space size. 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 `MC`^{2} algorithm could take advantage of them, as well.

7 Alternative Random-Sampling Strategies

To take a random sample, which in our case is a random lasso, `MC`^{2} performs a "uniform" random walk through the product Büchi automaton $B = B_S \times B_{\neg\varphi}$: one in which in order to decide which transition to take next, a fair, k-sided coin is tossed when a state of B is reached having k outgoing transitions. No attempt is made to bias the sampling towards accepting lassos, which is the notion of success for the Bernoulli random variable Z upon which `MC`^{2} is based. We are currently experimenting with alternative sampling strategies that favor accepting lassos.

[4] The principals in the Needham-Schroeder protocol use nonces—previously unused and unpredictable identifiers—to ensure secrecy.

Multi-lassos The *multi-lasso* sampling strategy ignores back-edges that do not lead to an accepting lasso if there are still forward edges to be explored. As shown below, this may have dramatic consequences.

In the case where the out-degree of B's states is nearly uniform, the sampling currently performed by $\texttt{MC}^2$ is biased toward shorter paths. To see this, consider for simplicity, the case where the out-degree is constant at $k > 1$. Then, the probability of a random lasso of length l is $(\frac{1}{k})^l$ and the shorter the lasso, the higher its probability. Thus, when S is *not* a model of φ, $\texttt{MC}^2$ is likely to first sample, and hence identify, a *shorter* counter-example sequence rather than a longer one. Given that shorter counter-examples are easier to decode and understand than longer ones, the advantage of this form of biased sampling becomes apparent.

On the other hand, one can construct an automaton that is adversarial to the type of sampling performed by $\texttt{MC}^2$. For example, consider the Büchi automaton B of Figure 2 consisting of a chain of $n+1$ states, such that for each state there is also a transition going back to the initial state. Furthermore, the only final state of B is the last state of the chain. Then there are $n+1$ lassos $l_0, \ldots, l_n$ in B, only one of which, l_n, is accepting. Moreover, according to Definition 6, the probability assigned to l_n is $1/2^n$, requiring $O(2^n)$ samples to be taken to sample l_n with high probability.

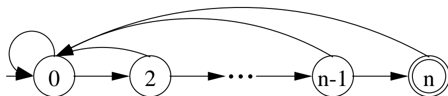

Fig. 2. Adversarial Büchi automaton B.

Interpreting automaton B of Figure 2 as the state-transition behavior of some system S, observe that B itself is not probabilistic even if the sampling performed on it by $\texttt{MC}^2$ is. In fact, it might even be the case that lasso l_n corresponds to a "normal" or likely behavioral pattern of S, making its detection essential. In this case, the adversarial nature of B is evident. Using a multi-lasso strategy however, dramatically increases the probability of l_n to 1, as the size of the multi-lasso space of B is 1.

Probabilistic systems In *probabilistic model checking*—see, for example, [29]—the state-transition behavior of a system S is prescribed by a probabilistic automaton such as a discrete Markov chain M. 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 $\texttt{MC}^2$ extends with little modification to the case of probabilistic model checking. Also, the example of Figure 2 becomes less adversarial as l_n would indeed in a probabilistic model be one of very low probability.

Input partitioning When the probabilities of outgoing transitions are not known in advance, it seems reasonable to assign a uniform probability to transitions involving internal nondeterminism. This justifies the use of a sampling strategy

based on uniform random walks for closed systems as discussed above. For open systems, however, assigning a uniform probability to transitions involving external nondeterminism seems to be less than optimal: in practice, an attacker might use the same input to trigger a faulty behavior of the system over and over again. Since the external probabilities are in general unknown, a reasonable sampling strategy for open systems would be to partition (or abstract) the input into equivalence classes that trigger essentially the same behavior, and randomly choose a representative of each class when generating successor states.

8 Related Work

There have been a number of prior proposals for randomized approaches to the model-checking problem. Like our $\texttt{MC}^2$ algorithm, the Lurch debugger [37, 23] 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 $\texttt{MC}^2$ 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; $\texttt{MC}^2$, on the other hand, is a Monte Carlo model checker for general LTL formulas.

In [7] 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 [29, 4], Probabilistic I/O Automata [42], and Probabilistic Automata [39]. Like $\texttt{MC}^2$, these model checkers return results of a statistical nature.

Stochastic modeling formalisms and logics are also considered in [46, 24, 40], who advocate an approach to the model-checking problem based on random sampling of execution paths and statistical hypothesis testing. In particular, [24] 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 $\texttt{MC}^2$ 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 [34, 22, 43]. In contrast, $\texttt{MC}^2$ performs random sampling of lassos for general LTL model checking.

In [35], Monte Carlo and abstract interpretation techniques are used to yield upper bounds on the probability of certain outcomes of programs whose inputs are divided into two classes: those that behave according to some fixed probability distribution and those considered nondeterministic.

9 Conclusions

We have presented $\texttt{MC}^2$, what we believe to be the first randomized, Monte Carlo decision procedure for classical temporal-logic model checking. Utilizing basic probability theory of geometric random variables, $\texttt{MC}^2$ 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. Benchmarks show that $\texttt{MC}^2$ is fast, memory-efficient, and scales extremely well.

In terms of ongoing and future work, we are implementing the alternative sampling strategies discussed in Section 7. Also, we are seeking to improve the time and space efficiency of our jMocha implementation of $\texttt{MC}^2$ by "compiling" it into a BDD representation. This involves encoding the current state, hash table, and guarded assignments of each atom in a reactive module as BDDs, and implementing the next-state computation and the containment (in the hash table) check as BDD operations.

As an open problem, it would be interesting to extend our techniques to the model-checking problem for branching-time temporal logics, such as CTL and the modal mu-calculus. This extension appears to be non-trivial since the idea of sampling accepting lassos in the product graph will no longer suffice.

Acknowledgments: We would like to thank Richard Karp and Michael Luby for illuminating discussions about Monte Carlo estimation techniques.

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. Henzinger. Computer-aided verification. An introduction to model building and model checking for concurrent systems. An online version is available at http://www-cad.eecs.berkeley.edu/ tah/CavBook/.
3. R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, July 1999.
4. 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.
5. J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra.* Elsevier Science Publishers B.V., Amsterdam, 2001.
6. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS*, pages 193–207. Springer, LNCS 1579, 1999.
7. 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.
8. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

9. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.
10. E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Form. Methods Syst. Des.*, 9(1-2):77–104, 1996.
11. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
12. 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.
13. 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.
14. P. Dagum and M. Luby. An optimal approximation algorithm for bayesian inference. *Artificial Intelligence*, 78:1–27, 1997.
15. 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.
16. E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Form. Methods Syst. Des.*, 9(1-2):105–131, 1996.
17. R. Gerth, R. Kuiper, W. Penczek, and D. Peled. A partial order approach to branching time model checking. *Information and Computation*, 1997.
18. 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.
19. P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of Second Workshop on Computer-Aided Verification*, 1990.
20. 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`.
21. V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. *Formal Methods in System Design*, 15(3):217–238, November 1999.
22. P. Haslum. Model checking by random walk. In *Proc. of 1999 ECSEL Workshop*, 1999.
23. 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.
24. 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.
25. G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. of the Second Spin Workshop*, pages 23–32, 1996.
26. M. Jerrum and A. Sinclair. Polynomial time approximation algorithms for the ising model. *SIAM Journal on Computing*, 22:1087–1116, 1993.
27. R. Karp and M. Luby. Monte carlo algorithms for planar multiterminal network reliability. *Journal of Complexity*, 2:45–64, 1985.
28. R. Karp, M. Luby, and N. Madras. Monte-Carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10:429–448, 1989.
29. 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.

30. G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, pages 131–133, 1995.
31. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166. Springer-Verlag, 1996.
32. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
33. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.
34. 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.
35. 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.
36. R. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
37. D. Owen, T. Menzies, M. Heimdahl, and J. Gao. Finding faults quickly in formal models using random search. In *Proc. of SEKE 2003*, 2003.
38. 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.
39. 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.
40. 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.
41. A. P. Sistla and E. A. Emerson. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32:733–749, 1985.
42. 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.
43. 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.
44. A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.
45. 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.
46. 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.