Monte Carlo Model Checking

Radu Grosu and Scott A. Smolka

Dept. of Computer Science, Stony Brook Univ., Stony Brook, NY, 11794, USA
{grosu, sas}@cs.sunysb.edu

Abstract. We present $\texttt{MC}^2$, what we believe to be the first randomized, Monte Carlo algorithm for temporal-logic model checking. Given a specification S of a finite-state system, an LTL formula φ, and parameters ϵ and δ, $\texttt{MC}^2$ takes $M = \ln(\delta)/\ln(1-\epsilon)$ 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 $L(B) = \emptyset$. Let p_Z be the expectation of an accepting lasso in B. 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 the probability of finding an accepting lasso through further sampling, under the assumption that $p_Z \geq \epsilon$, is less than δ. It does so in time $O(MD)$ and space $O(D)$, where D is B's recurrence diameter, using an optimal number of samples M. Our experimental results demonstrate that $\texttt{MC}^2$ is fast, memory-efficient, and scales extremely well.

1 Introduction

Model checking [7, 23], 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 of which is *state explosion*: the phenomenon where the size of a system's state space grows exponentially in the size of its specification. See, for example, [27], where it is shown that the problem is PSPACE-complete for LTL (Linear Temporal Logic). Over the past two decades, researchers have developed a plethora of techniques (heuristics) aimed at curtailing state explosion, including symbolic model checking, partial-order reduction methods, symmetry reduction, and bounded model checking. A comprehensive discourse on model checking, including a discussion of techniques for state explosion, can be found in [6].

We present in this paper an alternative approach to coping with state explosion based on the technique of *random sampling* by executing a random walk through the system's state transition graph. Such a technique was first advocated by West [31, 32] and Rudin [24] to find errors (safety violations) in communication protocols. We show how this this technique can be extended and formalized in the context of LTL model checking.

N. Halbwachs and L. Zuck (Eds.): TACAS 2005, LNCS 3440, pp. 271–286, 2005.

Our approach makes use of the following idea from the automata-theoretic technique of Vardi and Wolper [30] 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. Call a cycle reachable from an initial state of B a *lasso*, and say that a lasso is *accepting* if the cycle portion of the lasso contains a final state of B. The presence in B of an accepting lasso 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 MC^2 Monte Carlo model-checking algorithm. Underlying the execution of 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 a random walk in B, starting from an initial state and terminating at a cycle, is an accepting lasso. MC^2 takes $M = \ln(\delta)/\ln(1-\epsilon)$ such random walks through B, each of which can be understood as a random sample Z_i. The random walks are 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, MC^2 returns false with l as a witness. Otherwise, it returns true and reports that the probability of finding an accepting lasso through further sampling, under the assumption that $p_Z \geq \epsilon$, is less than δ.

The main features of our MC^2 algorithm are the following.

- To the best of our knowledge, MC^2 is the first randomized, Monte Carlo algorithm to be proposed in the literature for the classical problem of temporal-logic model checking.
- 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] MC^2 also delivers *quantitative* information about the model-checking problem. Should the random sampling performed by MC^2 not reveal an accepting lasso in $B = B_S \times B_{\neg\varphi}$, MC^2 returns true and reports that the probability of finding an accepting lasso through further sampling, under the assumption that $p_Z \geq \epsilon$, is less than δ.
- MC^2 is very efficient in both time and space. Its time complexity is $O(MD)$ and its space complexity is $O(D)$, where D is B's recurrence diameter. Moreover, the number of samples $M = \ln(\delta)/\ln(1-\epsilon)$ taken by MC^2 is optimal.
- Although we present MC^2 in the context of the classical LTL model-checking problem, the algorithm works with little modification on systems specified using stochastic modeling formalisms such as discrete-time Markov chains.
- We have implemented MC^2 in the context of the JMocha model checker for Reactive Modules [2]. Our experimental results demonstrate that MC^2 is

[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.

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 considers the requisite probability theory of geometric random variables and hypothesis testing. Section 3 presents MC^2, our Monte Carlo model-checking algorithm. Section 4 describes our JMOCHA implementation of MC^2. Section 5 summarizes our experimental results. Section 6 considers alternative random-sampling strategies to the one currently used by MC^2. Section 7 discusses related work. Section 8 contains our conclusions and directions for future work. Appendix A of [10] provides an overview of automata-theoretic LTL model checking.

2 Random Sampling and Hypothesis Testing

As we will show in Section 3, 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 $F(N) = 1 - \delta$ for *confidence ratio* δ yields:

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

which provides the number of attempts N needed to achieve success (find a counter-example) with probability $1 - \delta$.

In our case, p_Z is in general unknown. However, given an *error margin* ϵ and assuming that $p_Z \geq \epsilon$ we obtain that

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

and therefore that $\mathbf{Pr}[X \leq M] \geq \mathbf{Pr}[X \leq N] = 1 - \delta$. Summarizing:

$$p_Z \geq \epsilon \quad \Rightarrow \quad \mathbf{Pr}[X \leq M] \geq 1 - \delta \quad \textbf{where} \quad M = \ln(\delta) / \ln(1 - \epsilon) \qquad (1)$$

In equation 1 gives us the minimal number of attempts M needed to achieve success with confidence ratio δ, under the assumption that $p_Z \geq \epsilon$.

The standard way of discharging such an assumption is to use *statistical hypothesis testing* (see e.g. [21]). To understand how this technique works, consider the following example.

Example 1 (Fair versus biased coin). Suppose there are two coins in a hat. One is fair and the other is biased towards tails. The task is to randomly select one of them and determine which one it is. To do this, one can proceed as follows: (i) Define the *null hypothesis* H_0 as "the fair coin was selected"; (ii) Perform N trials noting each time whether a heads or tails occurred; (iii) If the number of heads is "low", reject H_0. Else, fail to reject H_0. Two types of errors can occur in this scenario as shown in the following table:

	H_0 is true	H_0 is false
Reject H_0	Type-I error (probability α)	Correct to reject H_0
Fail to reject H_0	Correct to fail to reject H_0	Type-II error (probability β)

A type-I error occurs when H_0 is rejected even though it is true and a type-II error occurs when H_0 is not rejected even though it is false. A type-I error can be thought of as a false positive in the setting of abstract interpretation, while a type-II error can be viewed as a false negative. The probability of a type-I error is denoted by α and that of a type-II error by β; common practice is to find appropriate bounds for each of these error probabilities.

In our case, H_0 is the assumption that $p_Z \geq \epsilon$. Rewriting inequation 1 with respect to H_0 we obtain:

$$\mathbf{Pr}[X \leq M \,|\, H_0] \geq 1 - \delta \tag{2}$$

We now perform M trials. If no counterexample is found, i.e. if $X > M$, we reject H_0. This may introduce a type-I error: H_0 may be true even though we did not find a counter-example. However, the probability of making this error is bounded by δ; this is shown in inequation 3 which is obtained by taking the complement of $X \leq M$ in inequation 2:

$$\mathbf{Pr}[X > M \,|\, H_0] < \delta \tag{3}$$

Because we seek to attain a one-sided error decision procedure, we do not consider type-II errors in our application of hypothesis testing: as soon as we find a counter-example, we stop sampling and decide (with probability 1) that $S \models \varphi$ is false. To estimate the error probability and obtain a corresponding bound on the probability β of a type-II error,[2] we would need to continue sampling no matter how early on in the sampling process the first counter-example is encountered.

Such an approach is put forth by us in [11] where we show how to compute an (ϵ, δ)-approximation $\widetilde{p}_Z$ of p_Z; i.e., $\widetilde{p}_Z$ is such that:

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

[2] A type-II error arises in our setting when $p_Z < \epsilon$ even though we find a counter-example within M samples, thereby leading us to believe incorrectly that $p_Z \geq \epsilon$. Given that p_Z represents the probability that an arbitrary run of S is a counter-example to φ, one could say that we were "fortunate" to find a counter-example in this many samples.

As shown in [11], this can be done in a number of samples that is optimal to within a constant factor by appealing to the optimal approximation algorithm (OAA) of [8].

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. The number of samples taken by $\texttt{MC}^2$ is therefore usually an order of magnitude smaller 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.

$\texttt{MC}^2$ returns false at the first sample corresponding to an accepting lasso; i.e., it's tolerance level for errors is one. Relaxing this condition would allow $\texttt{MC}^2$ to continue sampling until an upper bound U on the number of counter-examples sampled is reached. Such an approach is related to the statistical quality control process used in manufacturing, where a batch of N items is rejected when more than U of them are found to be defective out of M randomly and sequentially chosen samples. This process is known in the literature as a *single acceptance plan with curtailed sampling* [9]. The computation of M is considerably more involved in this case, as it depends on the cumulative distribution function of a random variable with a negative binomial distribution.

3 Monte Carlo Model-Checking Algorithm

In this section, we present our randomized, automata-theoretic approach to model checking based on the DDFS algorithm given in Appendix A of [10] and the theory of geometric random variables and hypothesis testing presented in Section 2. 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 Appendix A of [10], 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 1 (Lasso Sample Space). *A finite run $\sigma = s_0x_0 \dots s_nx_ns_{n+1}$ of a Büchi automaton $B = (\Sigma, Q, Q_0, \Delta, F)$, is called a* lasso *if $s_0 \dots s_n$ 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 *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 2 (Run Probability). *The* probability $\mathbf{Pr}[\sigma]$ *of a finite run $\sigma = s_0x_0 \dots s_{n-1}x_{n-1}s_n$ 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.

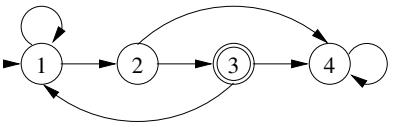

Fig. 1. Example lasso probability space

$\mathbf{Pr}[s_0] = k^{-1}$ *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 might explore uniformly successor states.

Example 2 (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, 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 3 (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 3 (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 $\texttt{MC}^2$, for the LTL model-checking problem. Its pseudo-code is as follows, where `rInit(B)=random(`$\mathtt{S}_0$`)`, `rNext(B,s)=t'`, $(\mathtt{s}, \alpha', \mathtt{t}') = \mathtt{random}(\{\tau \in \Delta \mid \exists \alpha, \mathtt{t}.\, \tau = (\mathtt{s}, \alpha, \mathtt{t})\})$ and `acc(s,B)=(s`$\in$`F)`. The main routine consists of three statements, the first of which uses inequation 1 of Section 2 to determine the value for M, given parameters ϵ and δ. The second statement is a for-loop that successively samples up to M lassos by calling the *random lasso* (`RL`) routine. If an accepting lasso l is found, $\texttt{MC}^2$ decides false and returns l as a counter-example. If no accepting lasso is found within M trials, $\texttt{MC}^2$ decides true, and reports that with probability less than δ, p_Z is greater 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`;

MC^2 algorithm
input: $\mathtt{B} = (\Sigma, \mathtt{Q}, \mathtt{Q}_0, \Delta, \mathtt{F})$; $0 < \epsilon < 1$; $0 < \delta < 1$.
output: Either (false, accepting lasso l) or (true, "$\mathbf{Pr}[X > M \mid H_0] < \delta$")

```
(1)  M := ln δ / ln(1 − ε);
(2)  for (i := 1; i ≤ M; i++) if (RL(B)==(1,l)) return (false,l);
(3)  return (true,"Pr[X > M | H0] < δ");
```

RL algorithm
input: Büchi automaton B;
output: Samples a RL l. Returns (1,l) if accepting; (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,lasso(HashTbl)) else return (0,Null);
```

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`.

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, $\mathbf{Pr}[X > M \mid H_0] < \delta$ where $M = \ln(\delta)/\ln(1-\epsilon)$ and $H_0 \equiv p_Z \geq \epsilon$.*

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 3 of Section 2.

MC^2 is very efficient in both time and space. The *recurrence diameter* of a Büchi automaton B is the longest loop-free path in B starting from an initial state.

Theorem 2 (MC^2 Complexity). *Let B be a Büchi automaton, D its recurrence diameter and $M = \ln(\delta)/\ln(1-\epsilon)$. Then* MC^2 *runs in time $O(MD)$ and uses $O(D)$ space. Moreover, M is optimal.*

Proof. The length of a lasso is bounded by D; the number of samples taken is bounded by M. That M is optimal follows from inequation 3, 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 $O(MD)$ time and $O(D)$ space. In the worst case, D is exponential in $|S| + |\varphi|$ and thus MC^2's asymptotic complexity would

match that of DDFS. In practice, however, we can expect MC^2 to perform better than this. For example, for the problem of N dining philosophers, our experimental results of Section 5 indicate that $D = O(N \cdot 1.4^N)$.

4 Implementation

We have implemented the DDFS and MC^2 algorithms as an extension to JMocha [2], 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 the composition of the system and formula modules 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, rlL, rlR, grL, grR, pc, and pr denote left and right request, left and right release, left and right grant, 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;
```

In JMocha, variables change their values in a sequence of rounds: an *initialization* round followed by *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 $\texttt{rqL?} \stackrel{\text{def}}{=} \texttt{rqL}' \neq \texttt{rqL}$ 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*. Parallel composition 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.

A feature of our jMocha implementation of $\texttt{MC}^2$ is that, given a Reactive module `M`, the next state along a random walk through M, $\texttt{s}' = \texttt{rNext(s,M)}$, is generated randomly both for the external variables `M.extl` and for the controlled variables `M.ctrl`. For the former, we randomly generate a state $\texttt{s.extl}'$ from the set of all input valuations `Q.M.extl`. For the latter, we proceed for each atom `A` in a linear order $\succ_{\texttt{M}}^{L}$ compatible with $\succ_{\texttt{M}}$ as follows. We first 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`. We then randomly generate a state $\texttt{s.ctrl}'$ from the set of all states 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.

5 Experimental Results

We compared the performance of $\texttt{MC}^2$ 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:

$$\texttt{DF} : \texttt{G}\,\neg\,(\texttt{pc}_1 = \texttt{wait}\,\&\,\ldots\,\&\,\texttt{pc}_\texttt{n} = \texttt{wait})$$
$$\texttt{SF} : \texttt{G}\,\texttt{F}\,(\texttt{pc}_1 = \texttt{eat})$$

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 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 $\texttt{MC}^2$ on finding counter-examples.

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

	DDFS		MC^2			
ph	time	entr	time	mxl	cxl	M
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		MC^2			
ph	time	entr	time	mxl	cxl	M
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 2. Deadlock and starvation freedom for fair asymmetric version

	DDFS		MC^2		
ph	time	entr	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
16	–	oom	3:03:40	7358	3144
20	–	oom	19:02:00	34158	14923

	DDFS		MC^2		
ph	time	entr	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
16	–	oom	2:50:50	6124	2983
20	–	oom	22:59:10	44559	17949

For the symmetric case, we chose $\delta = 10^{-1}$ and $\epsilon = 1.8 \cdot 10^{-3}$ which yields $M = 1257$. This number of samples proved sufficiently large in that for each instance of dining philosophers on which we ran our implementation of MC^2, a counter-example was detected. 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 length of the counter-example; `N` is the no. samples to find a counter-ex. As the data demonstrate, DDFS runs out of memory for 20 philosophers, while 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.

To avoid storing a large number of states in temporary variables, one might attempt to generate successor states one at a time (which is 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.

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

Table 3. Needham-Schroeder protocol

	DDFS		MC^2			
mr	time	entr	time	mxl	cxl	M
4	0.38	607	1.68	87	87	103
8	1.24	2527	11.3	208	65	697
16	5.87	13471	10.2	223	61	612
24	18.7	39007	3:06	280	44	12370
32	36.2	85279	2:54	269	63	11012

	DDFS		MC^2			
mr	time	entr	time	mxl	cxl	M
40	1:11	158431	1:46	325	117	7818
48	2:03	264607	1:45	232	25	6997
56	3:24	409951	6:54	278	133	28644
64	5:18	600607	7:12	347	32	29982
72	–	oom	11:53	336	63	43192

encode weak fairness. As in the symmetric case, we chose $\delta = 10^{-1}$ and $\epsilon = 1.8 \cdot 10^{-3}$. Our results are given in Table 2, where columns `mxl` and `avl` represent the maximum and average length of a sample, respectively.

The next model-checking benchmark we considered was the Needham-Schroeder public-key authentication protocol; first published in 1978 [22], 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 [17]. The following year, he showed how the flaw could be discovered mechanically by model checking [18].

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 in Table 3 where column `mr` represents 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 Table 1 for the symmetric (incorrect) version of dining philosophers.

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.

6 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 $B = B_S \times B_{\neg\varphi}$.

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

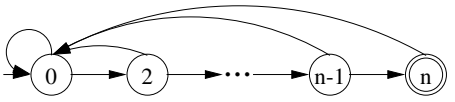

Fig. 2. Adversarial Büchi automaton B

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.

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 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, 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 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 2, 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.

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 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, [16]), the state-transition behavior of a system S is prescribed by a probabilistic automaton such as a discrete-time Markov chain (DTMC). 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 MC^2 extends with little modification to the case of LTL model

checking over DTMCs. 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.

7 Related Work

The Lurch debugger [14] performs random search on models of concurrent systems given as AND-OR graphs. Each iteration of the search function finds one global-state path, storing a hash value for each global state it encounters. The random search is terminated when the percentage of new states to old states reaches a "saturation point" or a user-defined limit on time or memory is reached. In [5] 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 [16, 4], Probabilistic I/O Automata [28], and Probabilistic Automata [25]. Examples logics treated by these model checkers include PCTL [12] and CSL [1]. Stochastic modeling formalisms and logics are also considered in [33, 15, 26]; these researchers, like us, advocate an approach to model checking based on random sampling of execution paths and hypothesis testing. The logics treated by these approaches, however, are restricted to time-bounded safety properties. Also, the number of samples taken by our algorithm—arrived at by appealing directly to the theory of geometric random variables—is optimal and therefore significantly smaller than the number of samples taken in [15].

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, 29]. In contrast, MC^2 performs random sampling of lassos for general LTL model checking. In [20], Monte Carlo and abstract interpretation techniques are used to analyze programs whose inputs are divided into two classes: those that behave according to some fixed probability distribution and those considered nondeterministic.

8 Conclusions

We have presented 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, 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 an optimal number of samples M. Benchmarks show that 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 6. Also, we are seeking to improve the time and space efficiency of our JMOCHA implementation of 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 Rajeev Alur, Javier Esparza, Richard Karp, Michael Luby, and Eugene Stark for helpful discussions. We are also grateful to the anonymous referees for their valuable comments.

References

1. A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous-time Markov chains, 1996.
2. 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.
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. 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.
6. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
7. 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.
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. A. J. Duncan. *Quality Control and Industrial Statistics*. Irwin-Dorsley, 1974.

10. R. Grosu and S. A. Smolka. Monte carlo model checking (extended version). In LNCS 3440 on SpringerLink. Springer-Verlag, 2005.
11. R. Grosu and S. A. Smolka. Quantitative model checking. In *First Intl. Symp. on Leveraging Applications of Formal Methods (Participants Proceedings)*, 2004. Also available from `http://www.cs.sunysb.edu/~sas/papers/GS04.pdf`.
12. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
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. 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.
17. G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, pages 131–133, 1995.
18. 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.
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. A. M. Mood, F.A. Graybill, and D.C. Boes. *Introduction to the Theory of Statistics*. McGraw-Hill Series in Probability and Statistics, 1974.
22. R. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
23. 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.
24. H. Rudin. Protocol development success stories: Part 1. In *Proc. 12th Int. Symp. on Protocol Specification, Testing and Verification*, pages 149–160. North Holland, 1992.
25. 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.
26. 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.
27. A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logic. *Journal of the ACM*, 32:733–749, 1985.

28. 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.
29. 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.
30. 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.
31. C. H. West. Protocol validation by random state exploration. In *Proc. Sixth IFIP WG 6.1 Int. Workshop on Protocol Specification, Testing, and Verification*. North Holland, 1986.
32. C. H. West. Protocol validation in complex systems. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 303–312. ACM Press, 1989.
33. 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.