STMC: Statistical Model Checker with Stratified and Antithetic Sampling

Nima Roohi[1], Yu Wang[2], Matthew West[3], Geir E. Dullerud[3], and Mahesh Viswanathan[3]

[1] University of California, San Diego, USA
nroohi@ucsd.edu
[2] Duke University, USA
yw354@duke.edu
[3] University of Illinois at Urbana-Champaign, USA
mwest,dullerud,vmahesh@illinois.edu

Abstract. `STMC` is a statistical model checker that uses antithetic and stratified sampling techniques to reduce the number of samples and, hence, the amount of time required before making a decision. The tool is capable of statistically verifying any *black-box* probabilistic system that `PRISM` can simulate, against probabilistic bounds on any property that `PRISM` can evaluate over individual executions of the system. We have evaluated our tool on many examples and compared it with both symbolic and statistical algorithms. When the number of strata is large, our algorithms reduced the number of samples 336% on average. Furthermore, being a statistical model checker makes `STMC` able to verify models that are well beyond the reach of current symbolic model checkers. On large systems (up to 10^{14} states) `STMC` was able to check 100% of benchmark systems, compared to existing methods which only succeeded on 14% of systems. The tool, installation instructions, benchmarks, and scripts for running the benchmarks are all available online as open source.

1 Introduction

Statistical model checking (SMC) plays an important role in verifying probabilistic temporal logics on cyber-physical systems [1, 16, 17]. In SMC, we treat the objective temporal specifications as statistical hypothesis, and infer their correctness with high confidence from samples of the systems. Compared to analytic approaches, statistical model checkers rely only on samples from the systems, and hence are more scalable to large real-world problems with complicated stochastic behavior [3, 7, 20].

To our knowledge, all existing SMC tools use independent samples. Admittedly, independent sampling is easy to implement, and it is the only option when the model is completely unknown. However, as shown recently in [27, 28], if the model is partially known, then we can exploit this knowledge to generate semantically negatively correlated samples to increase the sample efficiency in SMC.

Specifically, let us consider the core task of a statistical model checker $\mathcal{P}_{\leq p}\psi$, *i.e.*, to check if the satisfaction probability of a temporal logic specification ψ is no greater than a probability threshold $p \in [0,1]$. For simplicity, assume that the truth value of ψ can be determined by a finite prefix of the execution, and let $X_i \in \{0,1\}$ with $i = 1, \ldots, n$ be the truth value of ψ evaluated on n such (possibly dependent) prefixes. (Let 1 be true, and 0 be false.) Then, the satisfaction probability of ψ can be estimated by $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. One important measure that determines the accuracy of this estimate (and hence the sample efficiency) is its variance $\mathbf{Var}\bar{X} = \frac{1}{n^2}\sum_{i=1}^{n}\mathbf{Var}X_i + \frac{2}{n^2}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbf{Cov}(X_i, X_j)$. If the samples are i.i.d., then the covariance is 0; however, as can be seen from the above expression, the variance can be reduced if the samples are *semantically negatively correlated*, *i.e.*, $\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbf{Cov}(X_i, X_j) \leq 0$.

Currently, the only existing theory for generating such semantically negatively correlated samples is the *stratified and antithetic sampling* technique proposed in [27, 28] for discrete-time Markov chains (DTMC). In this work, we extend the technique to continuous-time Markov chains (CTMC), and implement the corresponding SMC algorithms by the tool `STMC`. The tool is evaluated on several case studies, some of which are beyond the capability of exhaustive model checking. The results show that the sample efficiency can be significantly improved by using semantically negatively correlated sampling, instead of independent sampling.

The rest of the paper is organized as follows. We present the problem formulation in Section 2 and our SMC algorithms using antithetic and stratified sampling in Section 3. The tool architecture and its internals are presented in Section 4. The evaluation of the tool on several benchmarks is presented in Section 5. Finally, we conclude this work in Section 6.

Related Work. Among the existing statistical model checkers, `PRISM` [4, 14], `MRMC` [12], `VESTA` [21], `YMER` [30], and `COSMOS` [2] only support independent sampling on DTMC, CTMC, or other more general probabilistic models. `PLASMA` [11] also supports importance sampling. In importance sampling, although samples may have different weights, they are still generated independently. To our knowledge, our tool `STMC` is the only existing statistical model checker that employs semantically negatively related sampling on DTMC and CTMC.

2 Problem Formulation

The tool `STMC` handles both discrete-time and continuous-time Markov chain. Below, we introduce the definitions for them.

Definition 1. A Labeled Continuous-Time Markov Chain *(CTMC) M is a tuple* $(\mathtt{S}, \mathtt{AP}, \mathtt{L}, \mathtt{T}, \mathtt{R}, \mathtt{s}^{\mathtt{init}})$*, where*

- $\mathtt{S}$ *is a finite set of* states*,*
- $\mathtt{AP}$ *is a finite set of* atomic propositions*,*
- $\mathtt{L} : \mathtt{S} \to 2^{\mathtt{AP}}$ *is the* labeling function*,*

– $\mathtt{T} : \mathtt{S} \times \mathtt{S} \to [0,1]$ *is the* transition probability function *with* $\sum_{s' \in S} \mathtt{T}(s,s') = 1$ *for any state* $s \in \mathtt{S}$.
– $\mathtt{R} : \mathtt{S} \to \mathbb{R}_+$ *is the* exit rate function*, and*
– $\mathtt{s}^{\mathtt{init}} \in \mathtt{S}$ *is the* initial state*.*

In addition, the tuple $(\mathtt{S}, \mathtt{AP}, \mathtt{L}, \mathtt{T}, \mathtt{s}^{\mathtt{init}})$*, is* a Labeled Discrete-Time Markov Chain *(DTMC).*

The DTMC generates a random path $\pi = s_0 \to s_1 \to \cdots$ by letting $s_0 = \mathtt{s}^{\mathtt{init}}$ and $s_{i+1} \sim \mathtt{T}(s_i, \cdot)$ for all $i \in \mathbb{N}$. Similarly, the CTMC generates a random path $\pi = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} s_n$ by letting $s_0 = \mathtt{s}^{\mathtt{init}}$, $s_{i+1} \sim \mathtt{T}(s_i, \cdot)$ and the time lapse t_i on s_i be from the exponential distribution with rate $\mathtt{R}(s_i)$ for all $i \in \mathbb{N}$. For a CTMC path, we denote the first state in π using $\mathtt{S}_0(\pi)$, the time of first transition in π using $\mathtt{R}_0(\pi)$, sum of all times in π using $\mathtt{Dur}(\pi)$, and π after removing its longest finite prefix π' such that $\mathtt{Dur}(\pi') \leq t$, using $\pi^{(t)}$. For example, if $\pi = s_0 \xrightarrow{1} s_1 \xrightarrow{2} s_2 \xrightarrow{3} s_3$, then $\pi^{(3)}$ is equal to $s_2 \xrightarrow{3} s_3$, and $\pi^{(4)}$ is equal to $s_2 \xrightarrow{2} s_3$.

STMC checks the satisfaction probability of Linear Temporal Logic (LTL) – *i.e.*, *probabilistic LTL* formulas on DTMCs and that of Metric Interval Temporal Logic formulas (MITL) – *i.e.*, *probabilistic MITL* formulas formulas on CTMCs. To simplify discussion, we introduce the following unified syntax for probabilistic LTL/MITL.

Definition 2 (Syntax). *A probabilistic LTL/MITL formula is defined by*

$$\varphi \ ::= \ \mathcal{P}_{\leq p}\psi \ \mid \ \mathcal{P}_{\geq p}\psi \qquad \psi \ ::= \ \top \ \mid \ \mathtt{AP} \ \mid \ \neg\psi \ \mid \ \psi \wedge \psi \ \mid \ \psi \mathcal{U}_I \psi$$

where $p \in (0,1)$ *is a probability threshold,* I *is a non-negative interval with extremities in* $\mathbb{N}$ *for LTL or in* $\mathbb{Q}$ *for MITL , and* $\mathtt{AP}$ *is an atomic proposition.*

Additional logic operators including *false* ($\bot$), *disjunction* ($\psi_1 \vee \psi_2$), *release* ($\psi_1 \mathcal{R}_I \psi_2$), *eventually* ($\Diamond_I \psi$), *always* ($\Box_I \psi$) and *next* ($\bigcirc \psi$) can be defined as syntactic sugars and they are all supported by STMC. The semantics for the temporal fraction is given as follows.

Definition 3 (Temporal Fraction Semantics). *Let* π *be a path of a CTMC (or DTMC) labeled by* $\mathtt{AP}$*, and* ψ *be an MITL (or LTL) formula over* $\mathtt{AP}$*. The path* π *satisfies* ψ*, denoted by* $\pi \models \psi$*, iff the following inductive rules hold.*

$$\begin{array}{lll}
\pi \models \top & \textit{iff} & \textit{true} \\
\pi \models \mathtt{a} & \textit{iff} & \mathtt{a} \in \mathtt{L}(\mathtt{S}_0(\pi)) \\
\pi \models \neg\psi & \textit{iff} & \pi \not\models \psi \\
\pi \models \psi_1 \wedge \psi_2 & \textit{iff} & \pi \models \psi_1 \textit{ and } \pi \models \psi_2 \\
\pi \models \psi_1 \mathcal{U}_I \psi_2 & \textit{iff} & \exists t_1 \in I \bullet \left(\pi^{(t_1)} \models \psi_2\right) \wedge \forall t_2 \in (0,t_1) \bullet \pi^{(t_2)} \models \psi_1
\end{array}$$

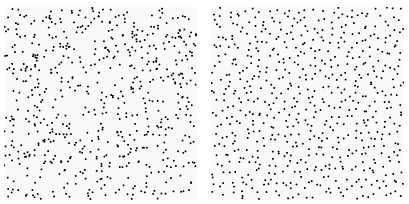
The probability operators from Definition 2 are interpreted as follows.

Definition 4 (Probability Operator Semantics). *Let M be a CTMC (or DTMC) and $\mathcal{P}_{\leq p}\psi$ be a probabilistic MITL (or probabilistic LTL) formula. The model M satisfies $\mathcal{P}_{\leq p}\psi$, denoted by $M \models \mathcal{P}_{\leq p}\psi$, iff the probability of a random path in M satisfying ψ is at most p. The semantics of $\mathcal{P}_{\geq p}\psi$ is defined similarly.*

Since we are using statistical techniques, we are not able to distinguish strict inequalities from non-strict ones. This is why we only have non-strict inequalities in Definition 2. In fact, *any* statistical algorithm assumes $M \models \mathcal{P}_{\leq p}\psi$ and $M \models \mathcal{P}_{\geq p}\psi$ are equivalent to $M \models \mathcal{P}_{<p}\psi$ and $M \models \mathcal{P}_{<p}\psi$, respectively.[4] Also, negation, conjunction and disjunction of probabilistic formulas are not included in the syntax from Definition 2 (as with `PRISM`). However, they can be handled by separately verifying each sub-formula. Finally, `PRISM` 4.5 only allows one temporal operator in a formula. On the other hand, our algorithms in `STMC` only assume that `PRISM` can evaluate LTL formulas on individual paths. Therefore, whenever `PRISM` lifts this limitation, our algorithms can be directly applied to the more powerful version.

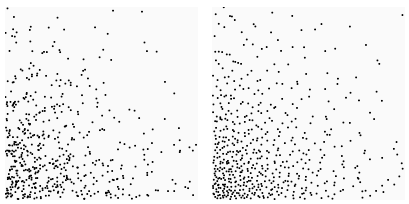
3 Stratified and Antithetic Sampling

Stratified and antithetic samplings are two approaches for generating negatively correlated random samples. When using stratified sampling to draw n samples from a distribution, we divide the support into sets with equal measure, and then draw one sample from each partition. When using antithetic sampling, a random seed is first drawn from $x \in [0, 1]$, and then two correlated samples are generated using x and $1 - x$, respectively. Figures 1 and 2 compare independent and stratified sampling for 625 samples that we drew from the joint distribution of two random variables. In Figure 1, each variable is uniformly distributed in $[0, 1]$, and in Figure 2, each variable is exponentially distributed with rate 3 (we only show samples that are within the unit square). It is clear that the stratified samples are (visually) better distributed in both figures.



(a) Independent **(b)** Stratified

Fig. 1: Uniform Distribution



(a) Independent **(b)** Stratified

Fig. 2: Exponential Distribution

[4] Many symbolic algorithms, including those in `PRISM`, make the same assumption as well. See our second remark at the end of Section 5.

Algorithm 1 Stratified Sampling for CTMC

```
// Take stratified samples and return fraction of samples that satisfy ψ.
// Param ψ is an LTL formula.
// Param strata_sizes is a non−empty list of positive integers.
function stratified_sampling(ψ, strata_sizes)
  val K     = strata_sizes.length      // Length of the list
  val N     = strata_sizes.product     // Product of elements in the list
  val paths = initialize N paths       // index starts at 0
  val evals = initialize N evaluators // incrementally evaluate ψ on paths
  // Evaluation in the condition of the while loop is performed by PRISM
  while(∃ j∈{0,...,N−1}, evals[j](path[j])='unknown')
    val π1 = random permutation of 0,1,...,N−1
    val π2 = random permutation of 0,1,...,N−1
    for(i ← 0,...,N−1)
      vars index1, index2 = π1[i], π2[i]
      for(s ← 0,...,K−1)
        val size = strata_sizes[s] // number of strata at step s
        vals offset1, offset2 = index1%size, index2%size
        index1, index2 /= size
        val rate = rate of last state in path[i]    // by PRISM
        val r1   = rnd(0,1) / size + offset1 / size // rnd(0,1) ∈ [0,1)
        val r2   = rnd(0,1) / size + offset2 / size
        val r3   = −ln(1−r2) / rate // stratified exponentially distributed
        Simulate one step in path[i] using r1 and r3 // by PRISM
  return number of paths that satisfy ψ / N
```

We have shown in [26] that by choosing a proper representation of a Markov chain, the stratified sampling technique can be applied to generate semantically negatively correlated sample paths. This technique reduces the sampling cost for statistically verifying temporal formulas. In the rest of this section, we list three algorithms: Stratified sampling of a CTMC, antithetic sampling of a CTMC, and stratified sequential probability ratio test for a CTMC. The antithetic variant can be easily obtained from the stratified one. Compared to our algorithms in [26], there are two main differences. First, we present these algorithms for CTMCs instead of DTMCs, as they are slightly more involved. Second, for the stratified sampling of a CTMC, our algorithm supports stratification over multiple steps directly.

Algorithm 1 shows the pseudo-code for stratified sampling of a CTMC; to obtain a stratified sampling algorithm for DTMC, we only need to remove π_2, index_2, offset_2, rate, r_2, and r_3. It takes two inputs – ψ, a temporal formula that we want to evaluate on every sampled path, and strata_sizes, the number of strata at every step. This is a non-empty list of positive integers. Let K be the length of this list, and N be the product of its elements. If the i^{th} item of the list is n then the number of strata at steps $i, i+K, i+2K, i+3K, \ldots$ must be n.[5] The algorithm simultaneously simulates N paths and terminates after

[5] The current version of PRISM only handles one initial state for simulation. Therefore, there will be no stratification for initializing paths.

Algorithm 2 Antithetic Sampling for CTMC

```
1  // Take antithetic samples and return fraction of samples that satisfy ψ.
2  // Param ψ is an LTL formula.
3  function antithetic_sampling(ψ)
4    val paths = initialize 2 paths        // index starts at 0
5    val evals = initialize 2 evaluators // incrementally evaluate ψ on paths
6    // Evaluation in the condition of the while loop is performed by PRISM
7    while(∃ j∈{0,...,1}, evals[j](path[j])='unknown')
8      vals rate1, rate2 = rates of last states in path[0] and path[1] // by PRISM
9      vals r1, r2 = rnd(0,1), rnd(0,1)     // one for destination and one for time
10     vals r3, r4 = 1−r1, 1−r2             // antithetic versions
11     vals r5, r6 = −ln(1−r2)/rate1, −ln(1−r4)/rate2//antithetic exp. distributed
12     Simulate one step in path[0] using r1 and r5    // by PRISM
13     Simulate one step in path[1] using r3 and r6    // by PRISM
14   return number of paths that satisfy ψ / 2
```

the value of ψ on all these paths are known. Inside the main loop, simulation is performed incrementally, K steps at a time. Random permutations π_1, π_2, and variables index_1, index_2 are used to make simulations of every K steps and random numbers r_1 and r_2 (defined later in the code) independent of each other. The number of strata at every step is an input to this algorithm. Using that number, variables offset_1 and offset_2 determine which strata we should use at step s. Finally, r_2 is a uniformly distributed stratified sample in $[0, 1)$. However, we need an exponentially distributed stratified sample, which is precisely what $-\ln(1-r_2)$ / rate gives us.

Algorithm 2 shows pseudo-code for the antithetic sampling of a CTMC (the same algorithm for a DTMC is obtained by removing rate_1, rate_2, r_2, r_4, r_5, and r_6). It takes only one parameter, ψ, which is a temporal formula that we want to evaluate on every sampled path. The algorithm simultaneously simulates two paths and terminates after the values of ψ on both paths are known. Inside the main loop, simulation is performed one step at a time. Also, in contrast to Algorithm 1, here we do not need random permutations. Simulation is performed by first sampling twice from a random variable uniformly distributed in $[0, 1)$, one for finding the next state and one for finding the time of transition. Then antithetic versions are stored in r_3 and r_4. Next, r_2 and r_4 are transformed into exponentially distributed stratified samples. Finally, PRISM is used to simulate each path for one more step.[6]

Finally, Algorithm 3 shows pseudo-code for statistical verification of a CTMC and DTMC using stratified samples. The antithetic version is obtained by replacing function call stratified_sampling (ψ,strata_sizes) with antithetic_sampling(ψ). Indeed, the algorithm is quite simple. It keeps sampling using Algorithm 1 and computes the average and variance of the values it receives until it satisfies a termination condition. Checking the termination conditions after every step sug-

[6] Strictly speaking, $1-r_4$ could be 0, which makes r_6 undefined. In an ideal situation, the probability of this happening is 0. However, since the implementation uses a *pseudo*-random number generator with a finite period, an extra check can be used to fix this issue.

Algorithm 3 Stratified Sequential Probability Ratio Test

```
1  // Verify P_{≤t}ψ using stratified sampling.
2  // Param t is the input threshold
3  // Param ψ is an LTL formula (non−probabilistic).
4  // Param strata_sizes is a non−empty list of positive integers.
5  // Param min_iter is the minimum number of iters. the algorithm should take.
6  // Param α is Type−I error probability (must satisfy 0 < α < 1/2).
7  // Param β is Type−II error probability (must satisfy 0 < β < 1/2).
8  // Param δ is half of the size of indifference region.
9  function stratified_SPRT(ℙ_{≤t}ψ, strata_sizes, min_iter, α, β, δ)
10   var iter = 1
11   var μ    = 0 // average of stratified_sampling return values
12   var σ    = 0 // standard deviation of stratified_sampling return values
13   while(true)
14     iter++
15     val x = stratified_sampling(ψ,strata_sizes)
16     update μ and σ using x // e.g. Welford's online algorithm [29]
17     if iter > min_iter then
18       if μ − t < −σ²/(2δ iter) ln (1−α)/β then return H_0
19       if μ − t >  σ²/(2δ iter) ln (1−β)/α then return H_1
```

gests using an online algorithm for computing the mean and variance of samples. We use Welford's online algorithm [29] in our implementation.

Finally, we recall the following results from [26] for Algorithms 1 to 3.

Theorem 5. *Let ψ be an LTL formula.*

1. *The output of Algorithms 1 and 2 have the same expected value as the probability of a random path satisfying ψ.*
2. *If ψ is of the form $\psi_1 \mathcal{U}_I \psi_2$, such that the set of states satisfying ψ_2 is a subset of the same set for ψ_1, then the satisfaction values of different paths simulated by Algorithms 1 and 2 are non-positively correlated.*

Proof. The first part holds as each individual stratified sample has the same probability distribution as an independent sample. The proof of the second part is similar to that of Theorem 1 in [28]. □

Theorem 6. *The sampling cost of Algorithm 3 is asymptotically no more than the sampling cost of SPRT [22] using i.i.d. samples.*

Proof. The proof follows from that of Theorem 6 in [28]. □

4 Tool Architecture

We have implemented our algorithms in `Scala` and published it under the GNU General Public License v3.0. The tool can be downloaded from https://github.com/nima-roohi/STMC/, where installation instructions, benchmarks, and scripts for running the benchmarks are located. We use `PRISM` to load models from files, simulate those models, and evaluate simulated paths against

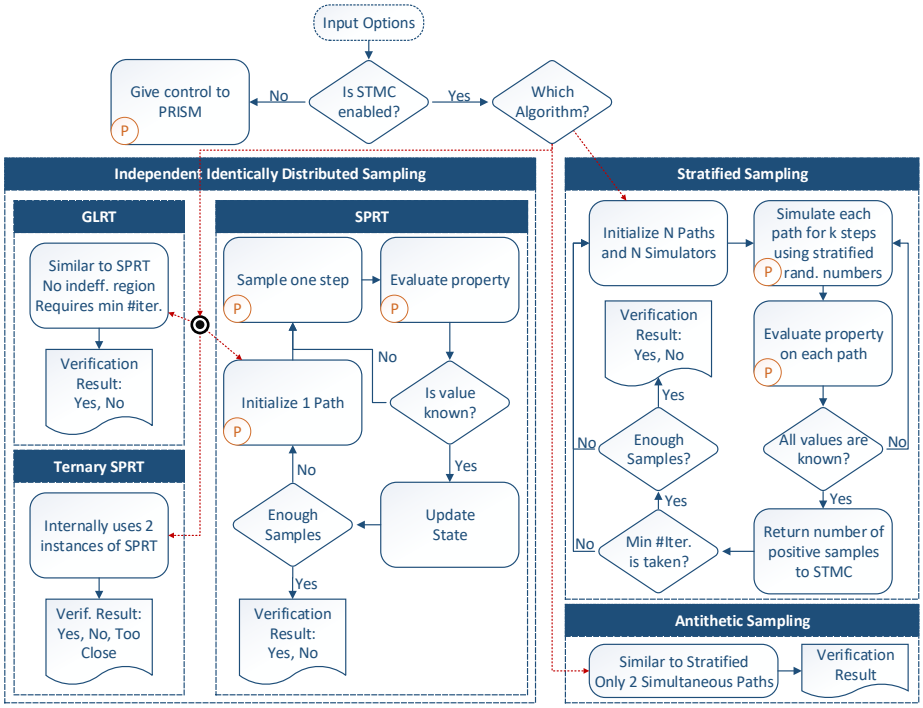


Fig. 3: Architecture of `STMC`. Boxes marked with letter 'P' use `PRISM` directly. N is the number of strata, K is the length of strata-size list (see option `strata_size` below).

non-probabilistic temporal properties. Therefore, `STMC` is capable of statistically verifying any model, as long as it can be understood by `PRISM`, and temporal properties can be evaluated on single executions of that model. Figure 3 shows `STMC` at a very high level. Boxes marked with 'P' are where we directly use `PRISM`. Also, since we did not use any native library in our implementation and since we also put the compiled version on the repository, whenever clients can run `PRISM`, they should be able to run `STMC` as well.

Executions of `STMC` are configured through different options/switches. The most basic options are `help`, which prints out a list of switches for both `STMC` and `PRISM`, and `stmc`, which enables the tool (without `stmc`, everything will be passed to `PRISM`, pretty much like `STMC` was not there in the first place). Statistical verification is enabled using option `sim`; it is always required when `stmc` is used. The sampling method is specified using option `smp_method` or `sm`. Possible values for the sampling method are `independent`, `antithetic`, and `stratified`. Using option `hyp_test_method` or `hm`, users also have to specify a hypothesis testing method that they would like to use. Supported values for this option are currently `SPRT`, `TSPRT`, `GLRT`, and `SSPRT`. `SPRT` is used for the sequential probability ratio test [22]. This algorithm has already been implemented in `PRISM` and in our experience it has a very similar performance to our implementation. We use our implementation for the next option, `TSPRT`. Sequen-

tial probability ratio test assumes that the actual probability is not within the δ-neighborhood of the input threshold. If this assumption is not satisfied, then the algorithm does not guarantee any error probability. `TSPRT`, which stands for Ternary SPRT, solves this problem by introducing a third possible answer: `TOO_CLOSE`. The algorithm was introduced in [31]. However, the paper does not coin any name for it. *Without* assuming that the actual probability is not within the δ-neighborhood of the input threshold, `TSPRT` guarantees Type-I and Type-II error probabilities are bounded by the input parameters α and β, respectively. Furthermore, it guarantees that if the actual probability and the input threshold are not δ-close, then the probability of returning `TOO_CLOSE` is less than another input parameter γ; we call this Type-III error probability.[7] The sequential probability ratio test was originally developed for simple hypotheses, and the test is not necessarily optimal when composite hypotheses are used [15]. To overcome this problem, the generalized likelihood ratio test (GLRT) was designed in [8]. The algorithm does not require an indifference region as an input parameter and provides guarantees on Type-I and Type-II error probabilities *asymptotically*. The main issue with this test is that since probabilistic error guarantees are asymptotic, for the test to perform reasonably well in practice (*i.e.*, respect the input error parameters), a correct minimum number of samples must be given as an extra input parameter. If this parameter is too large then the number of samples will be unnecessarily high, and if the parameter is too small then the actual error probability of the algorithm could be close to 0.5, even though the input error parameters are set to, for example, 10^{-7}. The last possible value for `hyp_test_method` is `SSPRT`, which stands for Stratified SPRT. This option should be used whenever stratified or antithetic samplings are desired.

When stratification is used, the number of strata should be specified using option `strata_size` or `ss`. It is a comma-separated list of positive integers. For example, $4, 4, 4, 4, 4, 4$ specifies 4 strata for six consecutive steps (4096 total), and 4096 specifies 4096 strata for every single step. Note that in both of these examples, stratified sampling simultaneously takes 4096 sample paths, which requires more memory. However, we saw in our experiments that for non-nested temporal formulas, at most two states of each path are stored into memory. Therefore, even larger strata sizes should be possible. This was the most challenging part of the implementation, because the simulator engine in `PRISM` is written assuming that paths are sampled one by one. However, if we followed the same approach in `STMC`, we had to store every random number that was previously generated, which increased the amount of memory used for simulation from $\mathcal{O}(1)$ to $\mathcal{O}(N \times L)$, where N is the number of strata and L is the maximum length of simulated paths. By simulating the paths simultaneously, we only use $\mathcal{O}(N)$ bytes of memory. Next, Type-I, Type-II, Type-III, and half of the size

[7] `PRISM` requires that the result of a test to be either a boolean value or an integer. Therefore, whenever this option is used, -1 means false, 1 means true, and 0 means the actual probability and input threshold are too close to make a decision.

of the indifference region are specified using `alpha`, `beta`,[8] `gamma` and `delta`, respectively (not every algorithm uses all of these parameters).

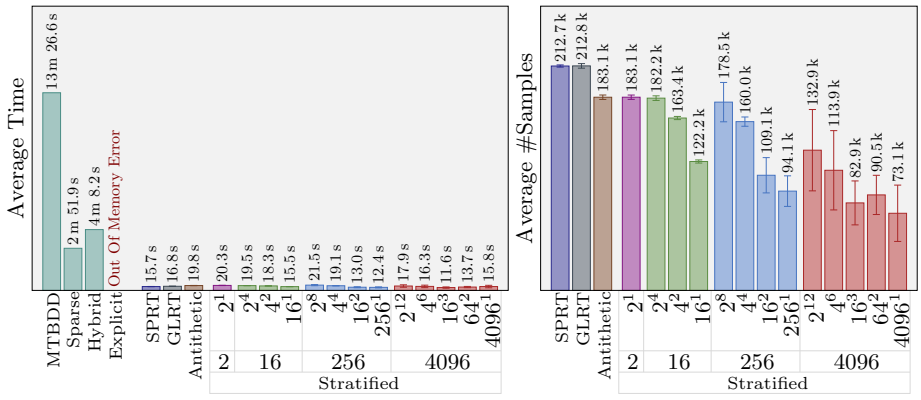
The current version of `STMC` also supports two additional options specifically designed for experimentally evaluating algorithms. The first option is `repeat`, which specifies the number of times the test should be repeated. This is useful when we evaluate a statistical algorithm experimentally, as the tool also computes the average and standard error for both time and number of samples each execution takes. The second option is `mt`. This instructs `STMC` to use as many processors as available to repeat the experiments. Note that `PRISM` is not thread-safe. Therefore, when `mt` is used, `STMC` creates different processes (and not threads) for each repeat. More explanations about these parameters, as well as examples for different scenarios, can be found in the tool repository.

5 Experimental Results

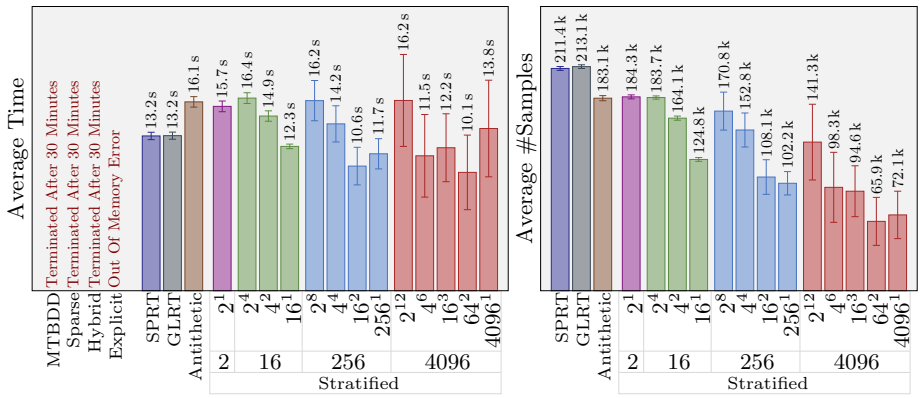
We evaluated our algorithms on nine different sets of examples. Each set contains four variations of the same problem with varying parameters of the model and, hence, various sizes, and each of those variations includes four symbolic tests as well as 16 statistical ones. Furthermore, we repeat each of the statistical tests 20 times, to compute 95% confidence intervals for time and number of samples taken by the statistical algorithms. This gives us a total of 720 tests and 11 664 runs to obtain experimental results for those tests. Regarding the stratified sampling, for each variation, we consider 13 settings in 4 groups. Each group uses a different number of strata: 2, 16, 256, and 4096. When the number of strata is more than 2, we also consider different possibilities for how to divide strata among different steps. For example, when 256 strata are used, 256^1 means every step has 256 strata, but different steps are independent of each other. On the other hand, 2^8 means every step has only two strata, but stratification is performed over every 8 consecutive steps. In other words, every possible list of length 8 in which elements are either $[0, 0.5)$ or $[0.5, 1)$ are considered. We present about a quarter of our results in the rest of this section. All the benchmark results are available at `STMC`'s webpage https://nima-roohi.github.io/STMC/#/benchmarks. Also, all the benchmark source files, along with scripts for running them, can be obtained from the tool's repository page https://github.com/nima-roohi/STMC/. To make our experiments faster, we ran all of our tests using four processes (using option '`-mt 4`'). Also, we used two different machines to increase speed. One of them is running Ubuntu 18.04 with an i7-8700 CPU 3.2GHz and 16GB memory, and the other one is running macOS Mojave with an i7 CPU 3.5GHz and 32GB memory.

NAND Multiplexing [19]. NAND multiplexing is a technique for constructing reliable computation from unreliable devices. As we move toward nanotechnology for device manufacturing, it is becoming more vital for architectures to

[8] To the best of our knowledge, `PRISM` always assumes $\alpha = \beta$.



(a) N: 90, K: 5, States: 113 384 792, Transitions: 180 005 807

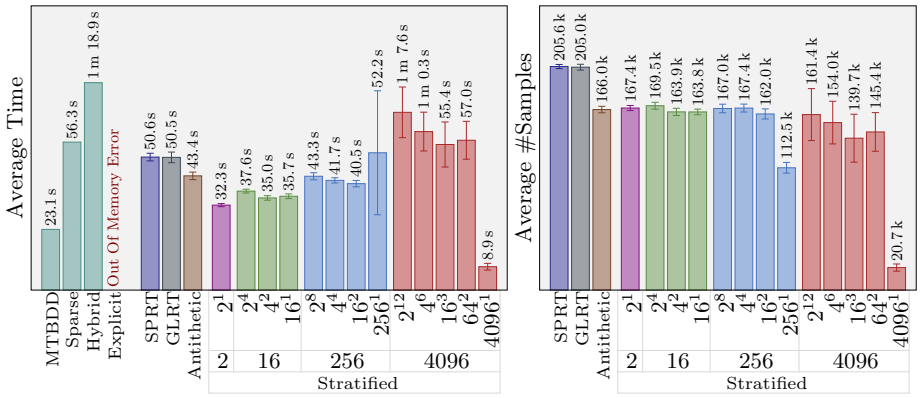


(b) N: 150, K: 11, States: 1 849 234 352, Transitions: 2 944 935 077

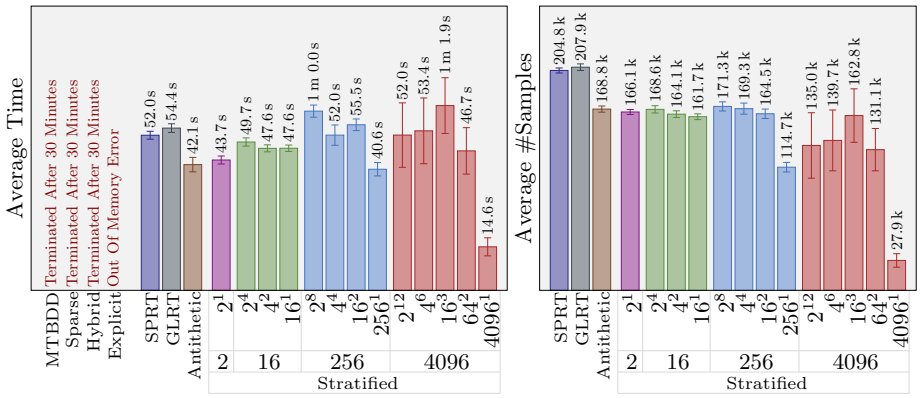
Fig. 4: NAND Multiplexing (DTMC)

be defect-tolerant. This is due to the fact that at nanoscale, devices are more likely to fail because of aging and transient faults. Micro-architects need to use redundancy to add defect tolerance to their designs. An implementation of this model can be found at https://www.prismmodelchecker.org/casestudies/nand.php. It has two parameters: N for the number of inputs in each bundle, and K for the number of restorative stages. Figure 4 shows the results for this set.

Embedded Control System [13, 18]. There are multiple subsystems in this example: an input processor (I) that reads and processes data from three sensors (S1, S2, S3), and the main processor (M) that polls those data and passes instructions to an output processor (O). This process occurs cyclically, and the main processor controls the length of this cycle with the help of a timer. The output processor controls two actuators (A1, A2) based on the instructions it receives. All three processors are connected using a bus. Multiple actuators and sensors are used for redundancy. The system will be functional if at least two



(a) `MAX_COUNT`: 10 000, States: 8 451 788, Transitions: 35 677 505

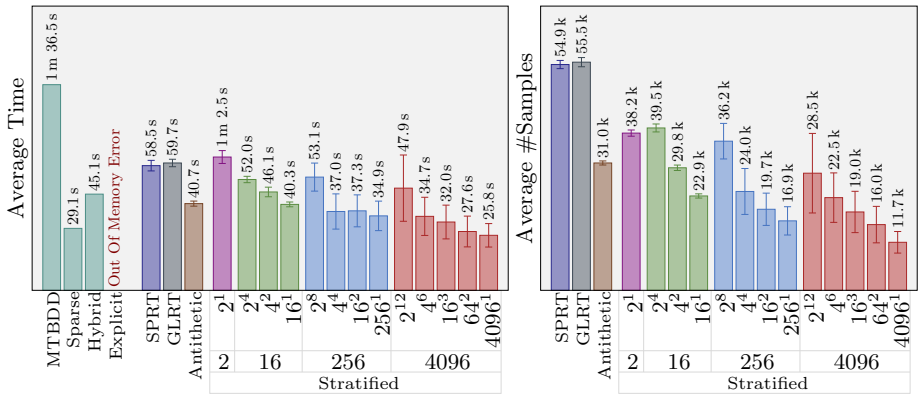


(b) `MAX_COUNT`: 1 000 000, States: about 845 017 880, Transitions: about 3 567 075 050

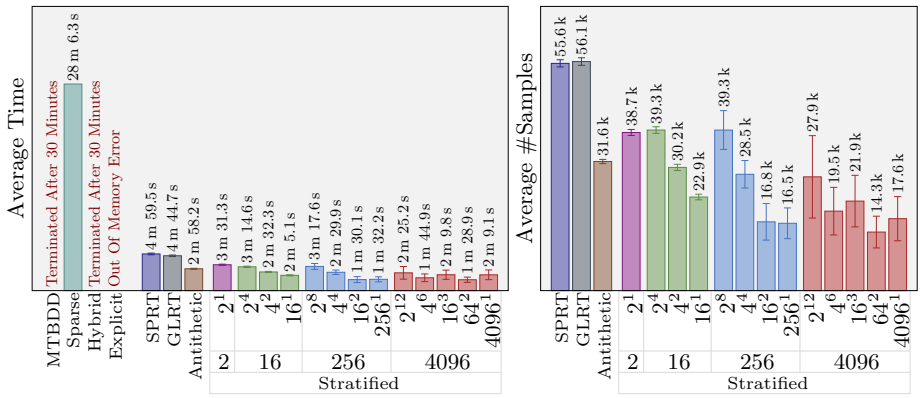
Fig. 5: Embedded Control System (CTMC)

sensors and one actuator are working. Otherwise, the input or output processor reports the failure to the main processor, and the system shuts down. Failure can also happen in the I/O processors. If this happens, then the main processor has to skip the current cycle. If M consecutively skips more than the input parameter `MAX_COUNT` cycles, then it assumes the failure is permanent and shuts down the system. Failure can also happen in the main processor, which immediately puts the system in shutdown. An implementation of this model can be found at https://www.prismmodelchecker.org/casestudies/embedded.php. Figure 5 shows the results for this set. Note that in Figure 5b, the size of the model is an estimation. This is because `PRISM` could not build the model after 90 minutes (using MTBDD as its engine).

Tandem Queueing Network [9]. This is a case study based on a simple tandem queueing network. It consists of an $M/Cox_2/1$-queue sequentially composed with an M/M/1-queue (a system having a single server, where a Poisson pro-



(a) c: 1 023, States: 2 096 128, Transitions: 7 328 771

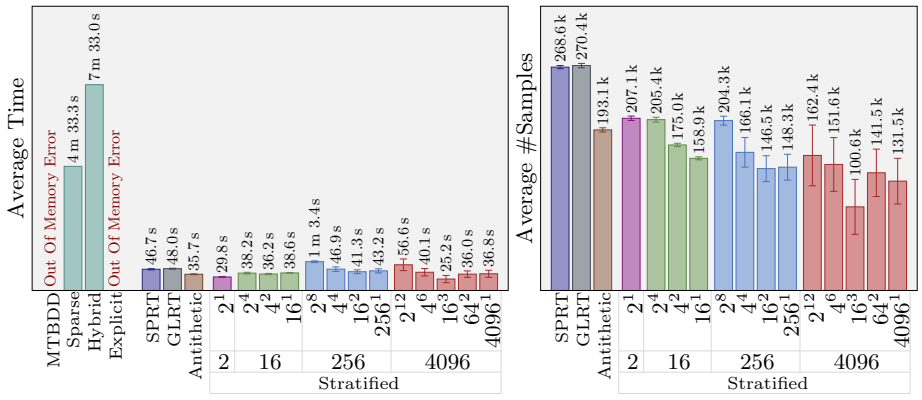


(b) c: 4 095, States: 33 550 336, Transitions: 117 395 459

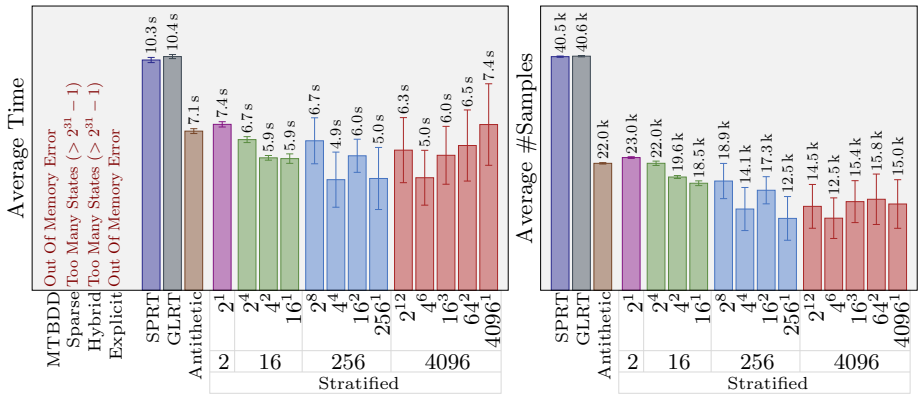
Fig. 6: Tandem Queueing Network (CTMC)

cess determines arrivals, and an exponential distribution determines job service times). Parameter c denotes the capacity of the queue. An implementation of this model can be found at https://www.prismmodelchecker.org/casestudies/tandem.php. Figure 6 shows the results for this set.

Flexible Manufacturing System [6]. There are three machines in this system: M_1, M_2, and M_3. Machine M_1 is triplicated and processes parts of type P_1, up to three at a time. Typically, machine M_2 processes parts of type P_2, one by one, but when no part of type P_2 is ready for the process, M_2 can also process parts of type P_3. After parts of types P_1 and P_2 are finished, machine M_3, which is duplicated, is used to put those parts together and create a new part of type P_{12}. Parts of every type can be shipped, and after this has happened the same number of rough parts enter the system. The time required to ship and replace parts is constant and independent of the number of parts involved. Parameter n denotes the total number of parts (could al-



(a) n: 10, States: 25 397 658, Transitions: 234 523 289



(b) n: 20, States: 8 831 321 730, Transitions: 94 963 796 928

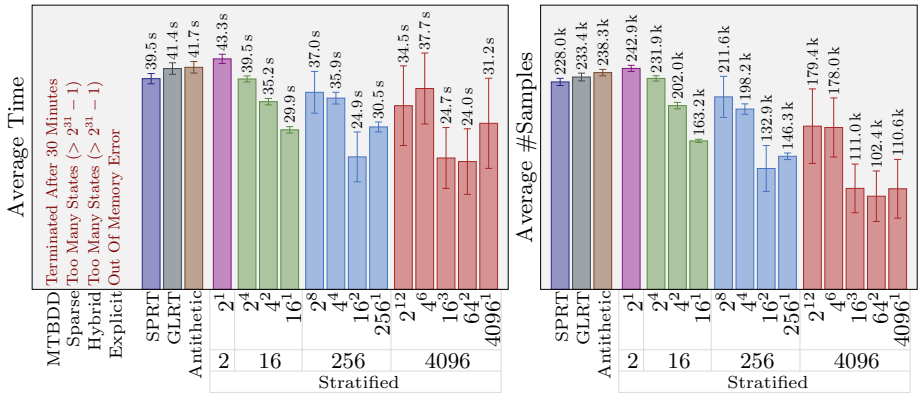
Fig. 7: Flexible Manufacturing System (CTMC)

ready be built or not) in the system. An implementation of this model is at https://www.prismmodelchecker.org/casestudies/fms.php. Figure 7 shows the results for this set.

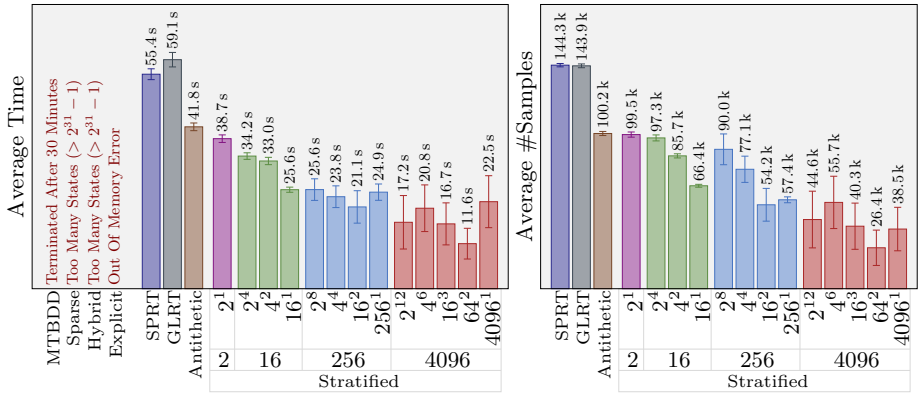
Simple Peer-To-Peer Protocol. This case study describes a simple peer-to-peer protocol based on BitTorrent. A file has been partitioned into K blocks, a client has all the blocks, and there are N other clients with no block. Each client can download a block from any other client. However, there could be at most four concurrent downloads for each block. An implementation of this model can be found at https://www.prismmodelchecker.org/casestudies/peer2peer.php. Figure 8 shows the results for this set.

A few notes about these results are in order.

– When the number of states increases, symbolic model checkers often terminates with one of the following errors: application ran out of memory, there



(a) `N`: 4, `K`: 8, States: 4 294 967 296, Transitions: 68 719 476 737



(b) `N`: 5, `K`: 8, States: 1 099 511 627 776, Transitions: 21 990 232 555 521

Fig. 8: Simple Peer-To-Peer Protocol (CTMC)

are too many states to start (*i.e.*, larger than $2^{31} - 1$) [9], or application ran over the 30 minutes time limit that we set in advance for it. For example, according to the second parts of Figures 4 to 8, out of 20 instances only the Sparse engine in Figure 6b survived the state space explosion, and even in that case, statistical algorithms terminated considerably faster.

– It is well known that symbolic model checkers do not scale well. Therefore, `PRISM` employs an iterative method to approximate probabilities. This approximation can be far from the actual probability, leading to incorrect model checking results. See [5] for examples.
– To make the configurations less in favor of statistical algorithms, we used small values for α, β, and δ in our benchmarks (between 0.0001 and 0.001). Also, we have estimated the actual probabilities using a symbolic model checker or

[9] In `PRISM` 4.5, when there are too many states, the symbolic engine automatically switches to MTBDD. Since we consider MTBDD separately, we count these cases as *Too Many States* error.

using a statistical algorithm in PRISM and set the threshold close to the actual probability. These settings cause the statistical algorithms to take more samples, which indeed makes it possible for us to observe the effect of antithetic and stratification on the number of samples. As a side effect, we did not observe any performance benefits of GLRT over SPRT.
- In many of our examples, the variance is particularly high when strata size is 4096. We believe this is because in our benchmarks, whenever 4096 strata are used, we set the minimum number of iterations to 2 (*i.e.*, 8192 samples). This means that when the average number of samples in our results is, for example, around 20 000, only 5 iterations have been taken on average, and every iteration adds or removes about 20% of the samples from the test.
- In general, the more strata we use, the greater reduction in the number of samples we observe. Also, the performance of antithetic sampling is similar to the case of using only two strata. Our best results are obtained when 4096^1 is used for the number of strata. For example, in Figure 5a, comparing SPRT and 4096^1 strata shows almost ten times reduction in the average number of samples. The tool's webpage contains an example in which stratification reduces variance to 0. This results in the termination of the algorithm immediately after a minimum number of samples have been taken, giving us 3 orders of magnitude reduction in the number of samples.

6 Conclusion

We presented our new tool called STMC for statistical model checking of discrete and continuous Markov chains. It uses antithetic and stratified sampling to improve the performance of a test. We evaluated our tool on hundreds of examples. Our experimental results show that our techniques can significantly reduce the number of samples and hence, the amount of time required for a test. When 4096^1 strata were used, our algorithms reduced the number of samples 336% on average. Furthermore, on large systems (up to 10^{14} states) STMC was able to check 100% of benchmark systems, compared to symbolic engines in PRISM which only succeeded on 14% of systems. We have implemented our tool in PRISM. Therefore, we can statistically verify any probabilistic model that can be simulated by PRISM against any property that it can evaluate on a single path. We published our tool under GNU General Public License v3.0, and the source code, installation instructions, benchmarks, scripts for running the benchmarks, and their results are all available online. We plan to extend our work in two directions. First, we would like to extend STMC to support other stratification-based algorithms. In particular, stratified sampling in model checking Markov decision processes, and temporal properties that are defined on the sequence of distributions generated by different types of Markov chains (see [23–25] for examples). Second, although our current extension of stratified sampling to multiple steps is very straight forward, we don't know if that is the best approach. We may even need to change the representation of the probability transition matrix to have a better sampling method.

References

1. Agha, G., Palmskog, K.: A Survey of Statistical Model Checking. ACM Trans. Model. Comput. Simul. **28**(1), 6:1–6:39 (2018)
2. Ballarini, P., Djafri, H., Duflot, M., Haddad, S., Pekergin, N.: COSMOS: A Statistical Model Checker for the Hybrid Automata Stochastic Logic. In: 2011 Eighth International Conference on Quantitative Evaluation of SysTems. pp. 143–144 (2011)
3. Barbot, B., Bérard, B., Duplouy, Y., Haddad, S.: Statistical Model-Checking for Autonomous Vehicle Safety Validation. In: SIA Simulation Numérique. Société des Ingénieurs de l'Automobile (2017)
4. Basu, S., Ghosh, A.P., He, R.: Approximate Model Checking of PCTL Involving Unbounded Path Properties. In: Formal Methods and Software Engineering. pp. 326–346. Springer Berlin Heidelberg (2009)
5. Bauer, M.S., Mathur, U., Chadha, R., Sistla, A.P., Viswanathan, M.: Exact quantitative probabilistic model checking through rational search. In: Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design. pp. 92–99. FMCAD '17, FMCAD Inc, Austin, TX (2017), http://dl.acm.org/citation.cfm?id=3168451.3168475
6. Ciardo, G., Trivedi, K.: A decomposition approach for stochastic reward net models. Performance Evaluation **18**(1), 37–59 (1993)
7. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., Sedwards, S.: Runtime Verification of Biological Systems. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change. pp. 388–404. Springer Berlin Heidelberg (2012)
8. Fan, J., Zhang, C., Zhang, J.: Generalized likelihood ratio statistics and wilks phenomenon. The Annals of Statistics **29**(1), 153–193 (2001), http://www.jstor.org/stable/2674021
9. Hermanns, H., Meyer-Kayser, J., Siegle, M.: Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In: Plateau, B., Stewart, W., Silva, M. (eds.) Proc. 3rd International Workshop on Numerical Solution of Markov Chains (NSMC'99). pp. 188–207. Prensas Universitarias de Zaragoza (1999)
10. Ibe, O., Trivedi, K.: Stochastic Petri net models of polling systems. IEEE Journal on Selected Areas in Communications **8**(9), 1649–1657 (1990)
11. Jegourel, C., Legay, A., Sedwards, S.: A Platform for High Performance Statistical Model Checking – PLASMA. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 498–503. Springer Berlin Heidelberg (2012)
12. Katoen, J., Khattri, M., Zapreevt, I.S.: A Markov reward model checker. In: Second International Conference on the Quantitative Evaluation of Systems (QEST'05). pp. 243–244 (2005)
13. Kwiatkowska, M., Norman, G., Parker, D.: Controller dependability analysis by probabilistic model checking. Control Engineering Practice **15**(11), 1427–1434 (2006)
14. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Computer Aided Verification, vol. 6806, pp. 585–591. Springer Berlin Heidelberg (2011)
15. Lai, T.L.: Sequential Analysis: Some Classical Problems and New Challenges. Statistica Sinica **11**(2), 303–351 (2001), http://www.jstor.org/stable/24306854
16. Larsen, K.G., Legay, A.: Statistical Model Checking: Past, Present, and Future. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques. pp. 3–15. Springer, Cham (2016)

17. Legay, A., Delahaye, B., Bensalem, S.: Statistical Model Checking: An Overview. In: Runtime Verification, vol. 6418, pp. 122–135. Springer Berlin Heidelberg (2010)
18. Muppala, J., Ciardo, G., Trivedi, K.: Stochastic reward nets for reliability prediction. Communications in Reliability, Maintainability and Serviceability **1**(2), 9–20 (July 1994)
19. Norman, G., Parker, D., Kwiatkowska, M., Shukla, S.: Evaluating the reliability of NAND multiplexing with PRISM. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **24**(10), 1629–1637 (2005)
20. Roohi, N., Wang, Y., West, M., Dullerud, G.E., Viswanathan, M.: Statistical verification of the Toyota powertrain control verification benchmark. In: 20th ACM International Conference on Hybrid Systems: Computation and Control (HSCC). pp. 65–70. ACM (2017)
21. Sen, K., Viswanathan, M., Agha, G.: VESTA: A statistical model-checker and analyzer for probabilistic systems. In: Second International Conference on the Quantitative Evaluation of Systems. pp. 251–252 (2005)
22. Wald, A.: Sequential tests of statistical hypotheses. Ann. Math. Statist. **16**(2), 117–186 (06 1945). https://doi.org/10.1214/aoms/1177731118, https://doi.org/10.1214/aoms/1177731118
23. Wang, Y., Roohi, N., West, M., Viswanathan, M., Dullerud, G.: A Mori-Zwanzig and MITL Based Approach to Statistical Verification of Continuous-time Dynamical Systems. International Federation of Automatic Control (IFAC PapersOnLine) **48**(27), 267–273 (2015)
24. Wang, Y., Roohi, N., West, M., Viswanathan, M., Dullerud, G.: Statistical Verification of Dynamical Systems using Set Oriented Methods. In: Hybrid Systems: Computation and Control (HSCC). pp. 169–178 (2015)
25. Wang, Y., Roohi, N., West, M., Viswanathan, M., Dullerud, G.: Verifying Continuous-time Stochastic Hybrid Systems via Mori-Zwanzig model reduction. In: 2016 IEEE 55th Conference on Decision and Control (CDC). pp. 3012–3017 (2016)
26. Wang, Y., Roohi, N., West, M., Viswanathan, M., Dullerud, G.: Statistical Verification of PCTL Using Antithetic and Stratified Samples. Formal Methods in System Design (2019)
27. Wang, Y., Roohi, N., West, M., Viswanathan, M., Dullerud, G.E.: Statistical verification of PCTL using stratified samples. In: 6th IFAC Conference on Analysis and Design of Hybrid Systems (ADHS), IFAC-PapersOnLine. vol. 51, pp. 85–90 (2018)
28. Wang, Y., Roohi, N., West, M., Viswanathan, M., Dullerud, G.E.: Statistical verification of PCTL using antithetic and stratified samples. Formal Methods in System Design (in press) (2019)
29. Welford, A.B.P., Welford, B.P.: Note on a method for calculating corrected sums of squares and products. Technometrics pp. 419–420 (1962)
30. Younes, H.L.S.: Ymer: A Statistical Model Checker. In: Computer Aided Verification. pp. 429–433. Springer Berlin Heidelberg (2005)
31. Younes, H.L.S.: Error control for probabilistic model checking. In: Emerson, E.A., Namjoshi, K.S. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 142–156. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)