How Fast and Fat Is Your Probabilistic Model Checker?
an experimental performance comparison[⋆]

David N. Jansen[1], Joost-Pieter Katoen[1,2], Marcel Oldenkamp[2], Mariëlle Stoelinga[2], and Ivan Zapreev[1,2]

[1] MOVES Group, RWTH, Aachen, Germany,
{David.Jansen@, katoen@cs., zapreevis@cs.}rwth-aachen.de
[2] FMT Group, University of Twente, Enschede, The Netherlands,
h.a.oldenkamp@student.utwente.nl, marielle@cs.utwente.nl

Abstract. This paper studies the efficiency of several probabilistic model checkers by comparing verification times and peak memory usage for a set of standard case studies. The study considers the model checkers ETMCC, MRMC, PRISM (sparse and hybrid mode), YMER and VESTA, and focuses on fully probabilistic systems. Several of our experiments show significantly different run times and memory consumptions between the tools—up to various orders of magnitude—without, however, indicating a clearly dominating tool. For statistical model checking YMER clearly prevails whereas for the numerical tools MRMC and PRISM (sparse) are rather close.

1 Introduction

Model checkers such as PRISM [33] (with about 4,000 downloads), MRMC [23], E⊢MC^2 [16], VESTA [34, 35], YMER [38], and APMC [26] support the verification of discrete- and continuous-time Markov chains. Their engines are based on combinations of numerical or simulation techniques for Markov chains and traditional CTL model-checking algorithms. Tools such as PRISM are relatively easy to use, have a graphical user interface and advanced built-in plot facilities. This allows researchers from various areas to apply probabilistic model checking. Applications range from areas such as randomized distributed algorithms to planning and AI, security [29], and even biological process modeling [27]. Probabilistic model checking engines have been integrated in existing tool chains for widely used formalisms such as stochastic Petri nets [9], Statemate [6], the stochastic process algebra PEPA [18], and a probabilistic variant of Promela [4].

This paper provides a comparative experimental study of a substantial set of probabilistic model checkers. The aim of this study is to get more insight into the strengths and weaknesses of the various tools, and to compare different model-checking techniques. We focus on fully probabilistic models, that is, finite-state discrete- and continuous-time Markov chains (DTMCs and CTMCs). We

[⋆] This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant numbers 642.000.505 (MOQS) and 612.000.311 (MC=MC); the EU under grant number IST-004527 (ARTIST2); the DFG/NWO bilateral cooperation programme under project number DN 62-600 (VOSS2); and by the DFG Research Training Group 1298 (AlgoSyn).

consider the temporal logics: probabilistic CTL (PCTL) [13] and its continuous-time variant CSL [3, 5]. These logics allow one to express constrained reachability probabilities, e. g., the probability to reach a goal state while visiting only legal states is at least 0.4567, and bounded versions thereof. In the discrete setting the bound is a number of steps while in the continuous case a time bound may be imposed on reaching the goal state. Finally, CSL allows for expressing steady-state properties such as: in the long run the probability to be in a goal state meets a certain bound. All these properties have been used in the experiments, as well as nested versions thereof and qualitative properties.

The experiments are focused on the verification time, i. e., the required time to verify a formula on a Markov chain, as well as peak memory usage, i. e., the maximal amount of memory needed during the verification. This was done for a set of five publicly available case studies, mostly examples that act as benchmarks for probabilistic model checking and that allow for varying state space sizes. Tools that were considered are E⊢MC^2, MRMC, PRISM, VESTA and YMER. All experiments were carried out on a standard PC, and care was taken that equivalent input models are used. Since models, properties, testing environment, and tool settings are all publicly available, all reported experiments are repeatable and verifiable. The number of experiments carried out is substantial, and each experiment is repeated several times. In total, about 15,000 verification runs have been considered. This paper presents a selection of the experiments from [30] and attempts to observe and explain relevant phenomena.

We found considerable differences in time and memory usage between the tools, due to variations in model checking techniques (statistical versus numerical), state space representation (MTBDDs versus sparse matrices or a combination) and implementation language (C/C++ versus Java). The tables in Sect. 6 show an overview of the results. In addition, we compared the user friendliness of the tools. Here PRISM is the clear winner.

Organisation of the paper. Section 2 briefly presents the tools and Sect. 3 the case studies we analyzed. In Sect. 4 we discuss the set up of our measurements. Then, we compare and analyze the results of our experiments in Sect. 5. Finally, Sect. 6 summarizes the conclusions and provides tool recommendations.

2 Tools

ETMCC. E⊢MC^2 [16] (version 1.4.2, 2001), also written ETMCC, is a prototype model checker for CTMCs. The tool is written in Java and uses sparse matrices to represent the state space.

MRMC. MRMC [23] (version 1.1.1b, March 2006) is a model checker for discrete-time and continuous-time Markov reward models. MRMC is a command-line tool, implemented in C, and represents the state space by sparse matrices.

PRISM. PRISM [25] (version 2.1, September 2004[3]) stands for Probabilistic Symbolic Model Checker. The user interface and parsers are written in Java; the core algorithms are mostly implemented in C++. For state space representation, PRISM uses a modified version of the CUDD package [37].

[3] This was the most recent version when we started our research. In the meantime, a newer version of PRISM has appeared.

PRISM offers a choice between two engines that use different data structures: a "sparse" and a "hybrid" engine, henceforth denoted as PRISM^S and PRISM^H. It is expected that PRISM^S is faster, whereas PRISM^H consumes less memory. Regardless of the engine, PRISM always generates an MTBDD to represent the transition matrix, and PRISM^S converts it to a sparse matrix, if necessary.

VESTA. VESTA [34] (version 2.0, 2005) is a Java-based tool for statistical analysis of probabilistic systems. It implements the statistical methods from [40, 35], based on Monte-Carlo simulation and statistical hypothesis testing [19].

YMER. YMER [38] (version 3.0, February 2005) is a command-line tool, written in C and C++, for verifying transient properties of CTMCs and generalizations. YMER implements statistical CSL model checking techniques [40], based on discrete event simulation [36] and acceptance sampling. It also supports numerical techniques, where the numerical engine for model checking CTMCs is adopted from PRIMS's hybrid engine.

Other tools. We have also considered other tools for our comparison, for example APMC [26], FHP-Murphi [31], Probverus [14]. We restricted ourselves to the above five because other tools did not support our models or logics or were not available in a stable version.

2.1 Languages

Input models. Most tools support both discrete- and continuous-time Markov chains. Support for discrete time is limited in E⊢MC^2; YMER only supports (a superset of) CTMCs. Some tools also recognize other input models (MDPs, reward models) not considered here. PRISM has its own modelling language: A system is described as the parallel composition of a set of *modules.* A module state is determined by a set of finite-range variables and its behaviour is given using a guarded-command notation. E⊢MC^2 and MRMC do not use a specific modeling language; instead, they accepts models in (a subset of) the *.tra*-format as e. g. generated by the stochastic process algebra tool TIPPtool [15] and Petri net tool DaNAMiCS [7]. The state labelling with atomic propositions has to be provided in a separate *.lab* file. We used a recently added feature of PRISM to generate these files directly from PRISM models. The language used by YMER is a subset of the PRISM language with a few slight syntactic differences. VESTA uses a Java-based language to specify models. A model description consists of sequential statements in combination with Java code. Each statement consists of a guard, rate and action. The language offers no explicit parallel composition.

Requirements. All tools support the logics PCTL for DTMCs, or CSL for CTMCs. The tools that support other models, of course, also know additional property languages. In addition, E⊢MC^2 supports aCSL, an action-based variant of CSL; and VESTA accepts requirements specified using QuaTEx [1].

3 Case studies

We selected five representative case studies, taken from literature on performance evaluation and probabilistic model checking. The selected studies represent a spectrum of applications, both distributed algorithms and performance models, and are of diverse natures. There are three discrete-time and two continuous-time

timing	study	min/max, param.	# states	# transitions
discrete	SLE	min, $n = 4, k = 2$	55	70
		max, $n = 8, k = 4$	458,847	524,382
	RDF	min, $n = 3$	770	2,845
		max, $n = 7$	5,454,562	44,070,594
	BDP	min, $m = 100$	101	202
		max, $m = 100{,}000$	100,001	200,002
continuous	TQN	min, $n = 2$	15	23
		max, $n = 1023$	2,096,128	7,328,771
	CPS	min, $n = 3$	36	84
		max, $n = 18$	7,077,888	69,599,232

Table 1. Minimal and maximal model sizes per case study

cases. For each case, we let the tools calculate the probability of some bounded and unbounded until properties, i. e. constrained reachability properties. They are the most important property type in the logic PCTL (and the only one that cannot be checked trivially). We also included a nested property (with multiple until operators) in a discrete-time case study. In the continuous-time case studies, we also checked for steady-state properties. The model types and the sizes of the smallest and largest models investigated are recorded in Table 1.[4]

Synchronous Leader Election (SLE). The Synchronous Leader Election protocol [21] solves the following problem: in a ring of n processors with synchronous unidirectional communication, the processors have to elect a unique leader by sending messages around the ring. The protocol proceeds in rounds. In a round, each processor (independently) chooses a random number from the set $\{1, \ldots, k\}$ as its id. Then, they pass their ids around the ring. If there is a unique id, then the processor with the largest unique id is elected leader; otherwise they begin a new round. We checked the SLE protocol for $n = 2$ with $k \in \{2, 4, 6, 8, 10, 12, 14, 16\}$ and $n = 4$ with $k \in \{2, 4\}$.

The protocol is used in several studies, e. g. [26, 12, 11]. We checked the properties: (1) eventually a leader is elected, i. e., $\mathcal{P}_{\geq 1}(\Diamond\, \mathit{elected})$, (2) the probability to elect a leader within 5 steps is ≥ 0.85, i. e., $\mathcal{P}_{\geq 0.85}(\Diamond^{\leq 5}\ \mathit{elected})$, and (3) the probability to elect a leader within 40 steps is ≥ 0.99, i. e., $\mathcal{P}_{\geq 0.99}(\Diamond^{\leq 40}\ \mathit{elected})$.

Randomized Dining Philosophers (RDP). In the Dining Philosophers problem [10], one assumes a round table with n philosophers who spend their lives just thinking and eating. There is a large plate of spaghetti in the center of the table, which is constantly refilled. Between each pair of philosophers lies a chopstick. Whenever a philosopher feels hungry, he can eat using the two chopsticks on his sides. [32] describes a distributed randomized algorithm to avoid deadlocks: A philosopher picks the two chopsticks in random order. If he can only get one chopstick, he gives up eating (but may become hungry again later).

For $n \in \{3, 4, 6, 7\}$ we checked the properties: (1) eventually some philosopher will eat, i. e., $\mathcal{P}_{\geq 1}(\Diamond\, \mathit{eat})$, and (2) the probability that some philosopher will eat within 20 steps is at least 0.9, i. e., $\mathcal{P}_{\geq 0.9}(\Diamond^{\leq 20}\ \mathit{eat})$.

Birth–Death Process (BDP). Birth–death processes [28, 22] are used in numerous fields, e. g. to model the growth of a population or queue size. States in

[4] Unfortunately we were not able to generate larger state spaces for the SLE case study due to an error obtained from the CUDD package.

a birth–death process are numbered by integers that denote the current population size n. An increase in size is denoted as "birth" whereas a decrease is denoted as "death." To get a finite Markov chain, we limited the maximum population size to a predetermined size m. The probability of birth decreases with the population size, until it is 0 when the maximum population is reached.

For $m \in \{100, 1000, 10000, 100000\}$ we checked the properties: (1) the probability to reach a quarter of the maximum population within $\frac{m}{2}$ steps is ≥ 0.9, i. e., $\mathcal{P}_{\geq 0.9}(\Diamond^{\leq \frac{m}{2}} (n = \frac{m}{4}))$, (2) eventually a population of 50 will be reached while the probability to reach a population of 70 within 100 steps never drops below 0.9, i. e., $\mathcal{P}_{\geq 0.8}(\mathcal{P}_{\geq 0.9}(\Diamond^{\leq 100}\ (n = 70))\,\mathcal{U}\ (n = 50))$, and (3) eventually the maximum population will be reached, i. e., $\mathcal{P}_{\geq 1}(\Diamond\ (n = m))$.

Tandem Queuing Network (TQN). The Tandem Queuing Network [17, 33] (see also [16, 39, 34]) consists of two queues of capacity n in sequence. Messages arrive at the first queue; when they get served, they are routed to the second queue, from where they leave the system. The message arrivals are exponentially distributed with rate $\lambda = 4n$. The server handles messages from the first queue according to a two-phase Coxian [8] distribution. The time between departures from the second queue is exponentially distributed with rate $\kappa = 4$.

For $n \in \{2, 10, 50, 100, 255, 511, 1023\}$ we checked: (1) in equilibrium, the TQN is full with probability < 0.01, i. e., $\mathcal{S}_{<0.01}(\mathit{full})$, (2) the TQN is full within 0.5 to 2 time units with probability < 0.1, i. e., $\mathcal{P}_{<0.1}(\Diamond^{[0.5,2]}\ \mathit{full})$, and (3) if the second queue is full, eventually a departure will happen, i. e., $\mathcal{P}_{\geq 1}(\mathit{snd}\,\mathcal{U}\ \mathit{sndn})$.

Cyclic Server Polling System (CPS). A cyclic polling system [20] consists of n stations and a server. Each station has a buffer with capacity 1 and the stations are attended by a single server in cyclic order. The server starts by polling the first station. If this station has a message in its buffer, the server serves it. Once the station has been served, or if its buffer was empty, the server moves to the next station cyclically. The polling and service times are exponentially distributed with rates $\gamma = 200$ and $\mu = 1$, respectively. The arrival rate of messages at each station is exponentially distributed with rate $\lambda = \mu/n$. Applications of this case study can be found in e. g. [38, 16, 34, 39].

For $n \in \{3, 6, 9, 12, 15, 16, 17, 18\}$ we checked properties like: (1) in the steady state, the first station is waiting for the server with probability < 0.2, i. e., $\mathcal{S}_{<0.2}(\mathit{busy}_1\ \wedge\ \neg \mathit{serve}_1)$, (2) the probability that the first station will be served within time interval $[40, 80]$ is ≤ 0.99, i. e., $\mathcal{P}_{\leq 0.99}(\Diamond^{[40,80]}\ \mathit{serve}_1)$, (3) if the first station is busy, the probability that it will be served within time t is ≥ 0.5 (for $t \in \{5, 10, 20, 40, 80\}$), i. e., $\mathit{busy}_1 \implies \mathcal{P}_{\geq 0.5}(\Diamond^{\leq t}\ \mathit{poll}_1)$, and (4) if the first station is busy, it will be served eventually, i. e., $\mathit{busy}_1 \implies \mathcal{P}_{\geq 1.0}(\Diamond\, \mathit{poll}_1)$.

4 Experimental setup

This section describes the details of our experiments measuring the verification time and peak memory usage of the various tools. To give our conclusions a solid scientific basis, the experiment design was guided by the following principles:

– *Repeatability and Verifiability:* Every one should be able to repeat and verify our experiments; this is achieved by the fact that our models, properties, scripts and tool settings are publicly available.

– *Statistical Significance:* This has been achieved by repeating experiments several times and computing the standard deviation.
– *Encapsulation:* Our experiments should measure what we claim to measure (i. e. model check times and memory usage), no other influences. This has been achieved by carefully measuring the time and memory usage of the processes (see below) and by using a dedicated machine, thus the effect of disturbing factors such as network traffic, background processes is avoided.

Moreover, we have considered the tools as black boxes. That is, we have executed the tools, but not changed their source code[5]. Also, we chose the verification parameters (e. g. the algorithm for solving matrix equations) to be the same across all tools. For details on the models and measurements, we refer to [30].

Software and hardware settings. All experiments were performed on a standard PC with an Intel® Pentium® 4 CPU 3.00 GHz processor and 2 GB of RAM. The operating system is SuSE Linux 9.1, because this is supported by all tools. Furthermore we ensured that the verification parameters and numerical solution methods of the tools match. For the numerical tools, e. g., the Jacobi method is used for solving systems of linear equations and the convergence accuracy ϵ is set to the default value 10^{-6}. For the statistical tools, we bound the probability of error (i. e. the chance of false negatives or positives) by $\alpha = \beta = 0.01$, which is the default setting for these tools, and half the width of the indifference region $\delta = 0.01$. The former agrees with possible choices of $\alpha = \beta$ from [39]. The choice of δ is somewhat arbitrary, and also taken from the literature.

Timing. In (probabilistic) model checking, two time factors are of interest: the *model construction time*, i. e. the time to build the internal representation from the input model, and the *model checking time*, i. e. the time to verify the property on the internal representation. We mainly focused on the bare model check time. One would often construct the model only once and then use it to verify multiple properties. In our comparison, we use the time as reported by the tools.

Memory usage. We measured the peak memory usage of the model checker, i. e. the amount of memory that is required for the verification problem at hand. More precisely, we recorded the virtual memory size (RAM + swap) of the entire process (which includes model construction). We did so by running a script in parallel to the model checker that took a sample every 100 msec. Although this sampling method is not perfect, it gives us the means to conduct uniform measurements on all tools, and it provides a reasonable indication of the memory consumption of each tool. A disadvantage is that this method does not work for very small experiments that are too quick. Other methods, such as profiling tools, are less suitable as they e. g., require tool modifications.

Data collection. All experiments and measurement procedures were automated using shell scripts. This enabled us to easily repeat experiments many times and collect data in a uniform way. An experiment consists of verifying one property on one particular model using one of the model checkers. The tools are restarted before each experiment; this prevents the interference of e. g., caching on the measurements. Each experiment was repeated 20 times, except that experiments for which a single run took more than 30 minutes were repeated only

[5] A minor exception is E⊢MC^2, where we added command line support to facilitate scripting.

three times. From the collected data, we calculated mean and standard deviation. The latter is determined using Student's t distribution, which takes the number of experiments into account. The maximal completion time for a single experiment was set to 24 hours, i. e., experiments that took longer were aborted. The verification time of these experiments is indicated in the results as ∞.

Model construction. The selected case studies were modeled using the model description language of each of the tools. For MRMC, E$\vdash$MC^2 and PRISM the models were readily available, viz., from the PRISM webpage or from the example models included in the tool distribution. Although the tools use different modeling languages, we require the models to be equivalent across all tools. Thanks to the export facility of PRISM version 3.0 beta1, models in the PRISM language can be exported to the input format of E$\vdash$MC^2 and MRMC. The YMER modeling language is almost identical to that of PRISM and only a few minor changes had to be made. The models for these four tools can thus safely be assumed to behave the same. The TQN and CPS case studies are provided in the standard distribution of the VESTA tool. Only for the BDP case study, a re-modeling effort was needed. We were not able to generate the models for the RDP and SLE case studies due to parsing problems of VESTA.

We attempted to generate models as large as possible by varying the model parameters. In addition to the RAM size, two factors restrict the model size: the size of the *.tra* files used by MRMC and E$\vdash$MC^2 is limited to a maximum of 2 GB In a few cases, we could not generate (and verify) our model as PRISM crashed due to a (known) problem of the CUDD package used for MTBDDs.

As MRMC and E$\vdash$MC^2 do not support a built-in modeling language, their overhead to generate a sparse matrix representation is low compared to the sparse matrix generation by PRISM. This aspect should be considered when interpreting the following experimental results.

5 Data and Analysis

5.1 Performance

The experimental results are discussed per type of formula, allowing us to compare phenomena across the various case studies. The results are presented by histograms where the x-axis indicates the model parameters that determine the state space size, and the y-axis indicates the verification time (in seconds) or the memory consumption (in KB). Note that the y-axis is log-scale. The legend of the plots is given by Fig. 1.

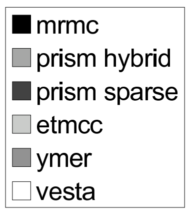
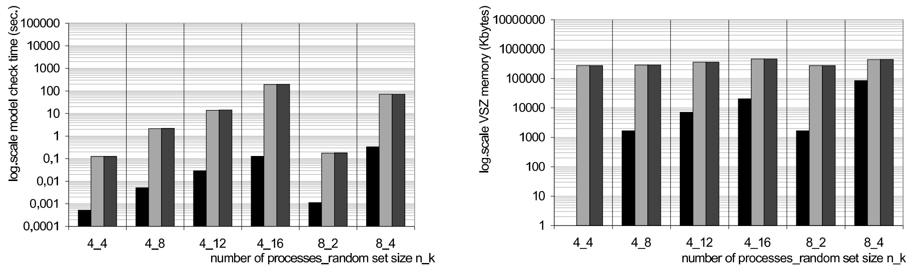


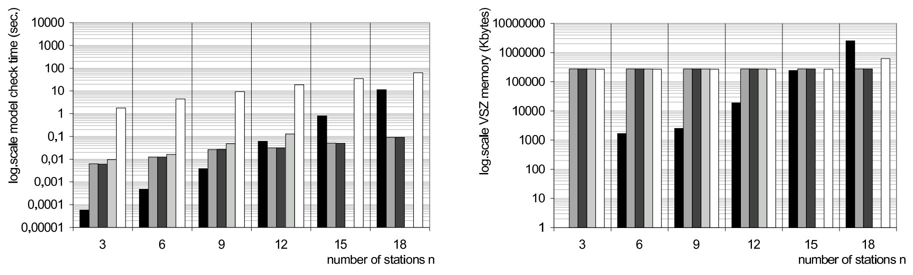
Fig. 1. The legend

Almost sure reachability properties. We first consider unbounded until formulas with probability bound ≥ 1. Figure 2 shows the verification time and memory usage for the SLE case study for various (n, k) pairs. (Recall that n is the number of nodes, and k the identity range.) As PRISM checks qualitative properties in a symbolic manner regardless whether it uses the sparse or hybrid engine, there is no difference in runtime nor in memory consumption between PRISM^S and PRISM^H. On increasing model parameters, the memory consumption of MRMC grows gradually (as expected) whereas for PRISM^S and PRISM^H only a slight increase is observed. This is due to the fact



(a) model check time (b) peak memory

Fig. 2. Synchronous leader election: $\mathcal{P}_{\geq 1}(\Diamond\, \mathit{elected})$



(a) model check time (b) peak memory

Fig. 3. Cyclic polling server: $\mathit{busy}_1 \implies \mathcal{P}_{\geq 1}(\Diamond\, \mathit{poll}_1)$

that PRISM requires a large base memory for the JVM, the CUDD package (around 40 MB) and the MTBDD it generates. The MTBDD for this case study is not very compact, as indicated by the following table:

(n, k)	$(4, 4)$	$(4, 8)$	$(4, 12)$	$(4, 16)$	$(8, 2)$	$(8, 4)$
MTBDD vertices	10K	1.6M	9M	27M	7K	10M
# states	0.8K	12K	62K	0.2M	2K	0.5M

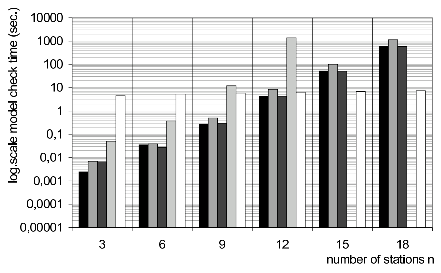
As a result, PRISM needs substantially more memory than MRMC and the verification times differ up to several orders of magnitude. (For the smallest two problem instantiations, the memory consumption for MRMC is unavailable as its verification times are negligible.)

The SLE case study suggests that memory consumption for PRISM^S and PRISM^H is highly influenced by the MTBDD size. This observation is also substantiated by the CPS case study, for which the MTBDD sizes just increase slightly on a growth of the state space size:

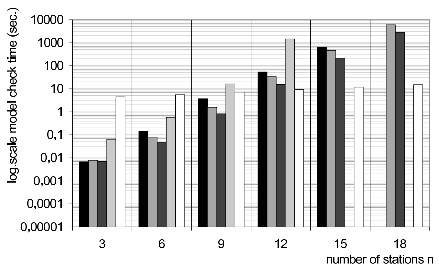
n	3	6	9	12	15	18
MTBDD vertices	112	367	765	1282	1942	2745
# states	36	0.6K	7K	74K	0.7M	7M

Observe that the MTBDD is very compact, e. g., the model of 7 million states only requires 2745 MTBDD vertices, much less than in the SLE case study.

Some experimental results for a reachability property of the CPS case study are summarized in Fig. 3. In contrast to the previous study, PRISM needs less memory than MRMC for large models due to the small MTBDD size. As before, there is no difference between PRISM^S and PRISM^H. For small models, MRMC is faster and less memory intensive, but for $n \geq 15$, it is outperformed



(a) model check time : $t = 5$



(b) model check time : $t = 80$

Fig. 4. Cyclic polling server: $busy_1 \implies \mathcal{P}_{\geq 0.5}(\Diamond^{\leq t}\ poll_1)$

by PRISM^S. This effect is to be expected to be more drastic for larger values of n as PRISM^S is able to check the CPS for $n > 18$ (roughly 26 M states) rather efficiently. As the file size of the *.tra* file for $n > 18$ exceeds 2 GB, we were unable to execute MRMC on it. For $n \geq 15$, E⊢MC^2 runs out of memory. The performance of E⊢MC^2 is worse than that of MRMC due to a less space-efficient sparse matrix representation, and the effect of the JVM. VESTA is about two orders of magnitude slower although—due to the use of Java—its memory usage is comparable to PRISM^S. The inefficiency of VESTA stems from the fact that it needs an excessive amount of sample paths to decide properties with bounds of the form ≥ 1, as shown in the following table:

n	3	6	9	12	15	18
# samples	34K	150K	395K	840K	1.6M	2.9M

Generally, statistical tools have difficulties to decide whether the probability of some property meets a bound if the actual probability and the bound are close. VESTA always gave the correct answer for these properties. For the BDP case study we experienced that for the property that almost surely eventually the population is maximal, VESTA reports an incorrect answer if the stopping probability—the likelihood that a sample path is stopped [35]—is not chosen appropriately. More precisely, if at some point during the simulation the stopping probability (in our case 0.05) is larger than that of reaching the state $n{=}m$ (in fact, a rare event), the sample path ends and it is concluded that $n{=}m$ is not reached. Re-simulation using a smaller stopping probability (e. g. 0.01) yields the correct answer. Note that YMER is not used here as it does not support unbounded reachability properties.

Bounded reachability properties. To show the effect of bounds, we consider a time-bounded variant of the property discussed before and observe what happens upon changing time bound t. Figure 4 depicts the verification times for the extreme bounds that we investigated in the CPS: $t{=}5$ and $t{=}80$, whereas Fig. 5 depicts the memory consumption for arbitrary t—the memory consumption does not

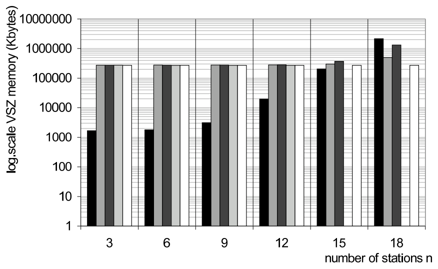
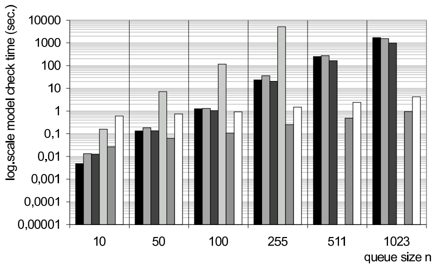
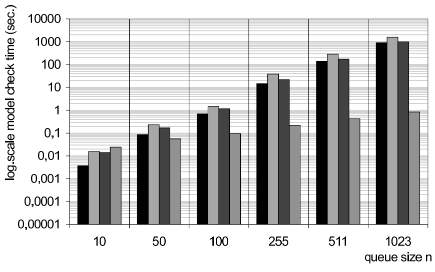


Fig. 5: Cyclic polling server, peak memory: $busy_1 \implies \mathcal{P}_{\geq 0.5}(\Diamond^{\leq t}\ poll_1)$



(a) model check time: $\mathcal{P}_{\leq 0.01}(\Diamond^{\leq 2}\ \mathit{full})$



(b) model check time: $\mathcal{P}_{\leq 0.01}(\Diamond^{[0.5,2]}\ \mathit{full})$

Fig. 6. Tandem queuing network: bounded reachability properties

depend on t. The verification time required by MRMC is heavily influenced by t, e. g., for n=15 the time for t=20 is about four times longer than t=5. This is not surprising, as the time complexity of the underlying algorithm is linear in t. From t=30 on, the verification time is almost constant, due to a built-in steady-state detection [24]. Besides, for t=80 and n=17, MRMC requires about 1700 seconds (not depicted), and we obtained a timer overflow for larger instantiations, i. e., the corresponding variable overflows. A similar behaviour is obtained for E⊢MC^2 but it runs out of memory rather quickly, as for simple reachability. PRISM^H is more efficient than PRISM^S due to the compact MTBDD (see previous case). As for MRMC, the verification time for PRISM^H and PRISM^S is linear in t, although this is less clear from the pictures due to the initial overhead of the MTBDD construction. A careful analysis of the logfiles reveals that the time *per iteration* is constant. Due to PRISM's steady-state detection, the verification time stops increasing around t=30. The verification time for VESTA for t=5 is rather constant as the number of samples (approx. 300,000) is more or less the same for each n. For t=80 the number of samples slightly increases (it raises from 0.2M for n=3 to about 1.1M for n=18). This explains the small increase in run time in Fig. 4(b). Unfortunately, VESTA gave wrong answers for low time bounds often: for $t = 5$, only 32.5 % of the answers were correct. Note that the property has also been checked by YMER, but as its run time is negligible—it immediately establishes that the initial state does not satisfy the premise of the implication—this is invisible in the figures. YMER thus has an "excellent" performance, but only checks the initial state whereas the other tools check *all* states. (VESTA also only provides answers for the initial state, but is unable to find the trivial satisfaction.)

Figures 6 and 7 show the results for checking a time-bounded property on the TQN case case study. YMER is for most cases much faster and smaller than all other tools. (For n=2 the verification time is too short to measure the memory consumption reliably.) As we have seen before, PRISM^H is more memory-efficient than PRISM^S, but the latter is faster. The memory usage of YMER is less than VESTA, and for

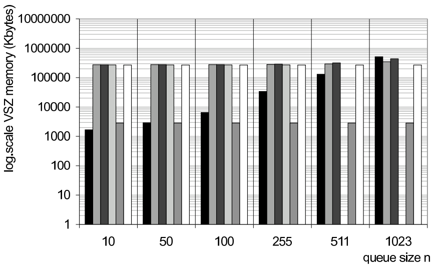
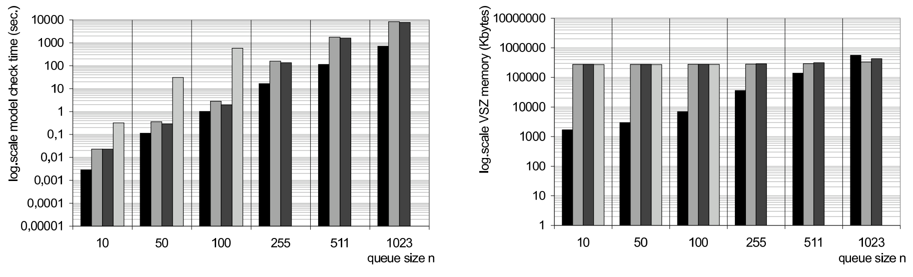


Fig. 7: Tandem queuing network, peak memory: $\mathcal{P}_{\leq 0.01}(\Diamond^{\leq 2}\ \mathit{full})$



(a) model check time (b) peak memory

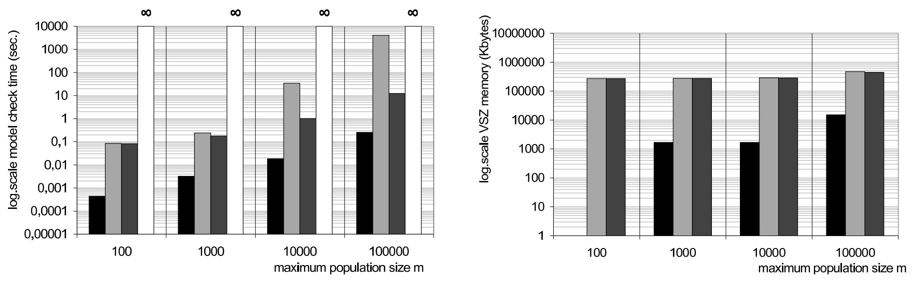
Fig. 8. Tandem queuing network : $\mathcal{S}_{>0.2}(\mathcal{P}_{>0.1}(\mathcal{X}\ snd))$

both simulation tools independent of the model size (as expected). As in the other case studies we see that due to the base memory overhead (JVM+CUDD) usage, the PRISM memory consumption is less dependent on the model size than MRMC, and E$\vdash$MC^2 is only able to handle relatively small models (up to few hundred thousands of states).

Figure 6(b) shows the timing for a bounded reachability property with both a positive lower and an upper bound. (E$\vdash$MC^2 and VESTA do not support these bounds.) To check this formula, a model checker will calculate two reachability probabilities in different Markov chains. The results are similar to the above, as expected: YMER is, for most cases, the fastest tool; its runtime depends less on the model size than for the other tools. MRMC is slightly faster than PRISM^S, which is slightly faster than PRISM^H. The fact that YMER is fast is also confirmed by checking such bounded property on the CPS case study, e. g. on n=16, YMER just needs 1.2 sec whereas PRISM^S and MRMC require about 1500 sec, and PRISM^H about 3000 sec.

Steady state properties. We only consider steady-state properties for CTMCs. The long-run operator for PCTL [2] is only supported by MRMC, and is therefore not used here. YMER and VESTA do not support steady-state properties, basically as it is unclear on when to stop the sample path generation in their applied techniques. Figure 8 shows the runtime and peak memory for a steady-state property in the TQN case study. The experiments show similar results as before. E$\vdash$MC^2 is the slowest tool and cannot handle large models (where $n > 100$). For the smaller models, the memory usage of PRISM is dominated by the overhead. For larger models, PRISM^S needs more memory than PRISM^H but is slightly faster. All experiments with steady-state formulas confirm our earlier observations: MRMC is faster and memory-wise more efficient than PRISM^S and PRISM^H, but for larger models, PRISM uses less memory than MRMC. The turn point, however, seems to occur at larger state spaces than experienced for the reachability properties.

Nested properties. We also checked the behaviour on nested quantitative reachability properties. Figure 9 shows the results of checking such property for the BDP case study. The tools check such nested formula in a bottom-up fashion, i. e., first the set of states satisfying the sub-formula is determined. The results are rather similar to the above findings. The MTBDD for the BDP case study is not very compact as the transition rates depend on the population size n, and as a result, most transition probabilities are distinct (resulting in many leaves in the MTBDD). As a result, MRMC outperforms PRISM^S and PRISM^H. Note



(a) model check time (b) peak memory

Fig. 9. Birth-death process : $\mathcal{P}_{\geq 0.8}(\mathcal{P}_{\geq 0.9}(\Diamond^{\leq 100}\ (n{=}70))\ \mathcal{U}\ (n = 50))$

however, that considered state spaces for this case study are relatively small which is favorable for MRMC. For all model instantiations, VESTA did not terminate simulation within 24 hours. We suggest as explanation that too many samples are required because the event n=70 is rather rare.

5.2 User friendliness

Our experiments also gave insight in the user friendliness of the probabilistic model checkers. As recognised by many people in the field, we find PRISM the most user friendly tool, having a reasonably powerful modeling language, a GUI and many additional features, such as the ability to plot the probability for different model parameter values. VESTA was less powerful in this respect. It does have a nice GUI, but lacks a parallel composition operator. Hence one needs to combine the various parallel components into a single model by hand, which is a very cumbersome and error-prone task. Also, we find VESTA's syntax and error messages not so intuitive. PRISM is able to generate files that are readable for E⊢MC^2, MRMC and YMER. Whereas E⊢MC^2 and MRMC allow one to read these files directly, YMER uses a slightly different syntax, so PRISM models have to be slightly transformed before being used by YMER. Without a GUI, all three tools are less intuitive to use than PRISM. On the other hand, MRMC is more appropriate as back-end verification engine as it has a simple input format.

The following table summarizes the results. Here, ++ is the best, −− is the worst, and 0 is neutral.

	E⊢MC^2	MRMC	PRISM	YMER	VESTA
ease of modeling	++ [a]	++ [a]	++	+	−−
ease of use	+	0/+	++	0	+

[a] Exploiting the modeling facilities of PRISM (or TIPPtool).

6 Conclusion

We presented a performance comparison of five probabilistic model checkers. By ensuring that our experiments are repeatable, verifiable, statistically significant and free from external influences, our findings are based on a solid methodology.

From our experiments, we conclude that YMER is by far the fastest tool. Also, its memory usage is remarkably constant, hardly varying with the model size. Unfortunately, YMER only supports bounded and interval until formulas. Also, as statistical tool, YMER may report the wrong answer, and has done so during our experiments (in a few cases, as expected). In particular, YMER outperforms the other statistical model checker VESTA: VESTA's memory consumption is also rather constant, but more in the order PRISM's memory usage. However, its runtime varies a lot. For certain nested properties we checked, VESTA did not terminate within 24 h, even on a model with 100 states only.

As expected, PRISM^S is usually faster than PRISM^H at the cost of substantially greater memory usage. E⊢MC^2 performs the worst in terms of memory, and frequently was unable to check models that were easy for the other tools.

For models up to a few million states, MRMC mostly performs better than PRISM^S both in time (although sparse matrix generation takes negligible time in MRMC compared to PRISM) and memory. This is mainly due to the overhead for MTBDD generation in PRISM. On larger models, PRISM^S and PRISM^H perform better. This effect is more apparent whenever the MTBDD representation is compact. As expected, PRISM^S is often faster than PRISM^H, but uses more memory. The results are summarized in the following tables.

speed	E⊢MC^2	MRMC	PRISM^S	PRISM^H	YMER	VESTA
steady state	–	++	+	0/+ [a]	N/A	N/A
bounded until	–	+ [b]	+/++	0/+ [a]	++	+
unbounded until	–	+ [b]	+/++	+/++ [a]	N/A	–/0
nested	–	++	+	0/+ [a]	N/A [c]	– – [d]

[a] The time heavily depends on the MTBDD size.
[b] MRMC was faster in most cases, PRISM^S on larger models.
[c] The property contained operators not supported by YMER.
[d] Based on one property, for which VESTA did not terminate.

memory	E⊢MC^2	MRMC	PRISM^S	PRISM^H	YMER	VESTA
steady state	–	+ [a]	+	+/++ [a] [b]	N/A	N/A
bounded until	–	+ [a]	+	+/++ [a] [b]	++	+ [c]
unbounded until	–	+ [a]	+/++	+/++ [a] [b]	N/A	0/+ [c]
nested	–	+ [a]	+	+/++ [a] [b]	N/A	N/A [d]

[a] MRMC used least memory in most cases. For larger models PRISM^S was between MRMC and PRISM^H, and PRISM^H was the best.
[b] The MTBDD size varied much with the case study.
[c] Fairly constant; inefficient for small models, efficient for large ones.
[d] Based on one property, for which VESTA did not terminate.

Recommendations. Based on our experience, we have the following suggestions for improving the tools. For YMER, it would be very useful if it supported more CSL/PCTL operators, so that its "slim and fast" engine becomes applicable to a wider class of model checking problems. Also, it would be nice for YMER to use exactly the same syntax as PRISM, improving the tool interoperability. For VESTA, we suggest to improve its running time. Also, its applicability would be enlarged by improving the modeling language, by either adding a parallel operator, or by supporting a modeling language similar to PRISM's. For PRISM, a tight connection with YMER could be of relevance — ideally, a user would call the YMER model checker by pressing a single button. For MRMC, we suggest to improve the performance for larger models.

Acknowledgement. We would like to thank Dave Parker and Gethin Norman for providing us with valuable feedback on a draft version of this paper.

References

1. Agha, G., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. *ENTCS* **153**(2) (2006) 213–239
2. Andova, S., Hermanns, H., Katoen, J.-P.: Discrete-time rewards model-checked. In Larsen, K. G., Niebert, P., eds.: *Formal Modeling and Analysis of Timed Systems: FORMATS. LNCS*, Vol. 2791, Berlin, Springer (2003) 88–104
3. Aziz, A., Sanwal, K., Singhal, V., Brayton, R. K.: Verifying continuous time Markov chains. In Alur, R., Henzinger, T. A., eds.: *Computer Aided Verification (CAV). LNCS*, Vol. 1102, Springer (1996) 269–276
4. Baier, C., Ciesinski, F., Größer, M.: ProbMela and verification of Markov decision processes. *SIGMETRICS Perform. Eval. Rev.* **32**(4) (2005) 22–27
5. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time Markov chains. *IEEE Trans. on Softw. Eng.* **29**(6) (2003) 524–541
6. Bode, E., Herbstritt, M., Hermanns, H., Johr, S., Peikenkamp, T., Pulungan, R., Wimmer, R., Becker, B.: Compositional Performability Evaluation for STATEMATE. In: *Quantitative Evaluation of Systems: QEST*, IEEE CS (2006) 167–178
7. Changuion, B., Davies, I., Nelte, M.: DaNAMiCS: a Petri net editor. http://www.cs.uct.ac.za/Research/DNA/microweb/danamics/DNAFrameH.html (2007)
8. Cox, D. R.: A use of complex probabilities in the theory of stochastic processes. *Proc. Cambridge Philosophical Society* **51** (1955) 313–319
9. D'Aprile, D., Donatelli, S., Sproston, J.: CSL model checking for the GreatSPN tool. In Aykanat, C., Dayar, T., Körpeoğlu, İ., eds.: *Computer and Information Sciences: ISCIS. LNCS*, Vol. 3280, Berlin, Springer (2004) 543–552
10. Dijkstra, E. W.: Hierarchical ordering of sequential processes. *Acta Informatica* **1** (1971) 115–138
11. Fokkink, W., Pang, J.: Simplifying Itai-Rodeh leader election for anonymous rings. *ENTCS* **128**(6) (2005) 53–68
12. Gupta, R., Smolka, S. A., Bhaskar, S.: On randomization in sequential and distributed algorithms. *ACM Comput. Surv.* **26**(1) (1994) 7–86
13. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. *Formal Aspects of Computing* **6**(5) (1994) 512–535
14. Hartonas-Garmhausen, V., Campos, S., Clarke, E. M.: ProbVerus: probabilistic symbolic model checking. In: *Formal Methods for Real-Time and Probabilistic Systems. LNCS*, Vol. 1601, Berlin, Springer (1999) 96–110
15. Hermanns, H., Herzog, U., Klehmet, U., Mertsiotakis, V., Siegle, M.: Compositional performance modelling with the TIPPtool. *Performance Evaluation* **39**(1-4) (2000) 5–35
16. Hermanns, H., Katoen, J.-P., Meyer-Kayser, J., Siegle, M.: A Markov chain model checker. In Graf, S., et al., eds.: *Tools and Algorithms for the Construction and Analysis of Systems: TACAS. LNCS*, Vol. 1785, Berlin, Springer (2000) 347–362
17. 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. J., Silva, M., eds.: *Num. Sol. of Markov Chains*, Zaragoza, Prensas Universitarias (1999) 188–207
18. Hillston, J.: *A Compositional Approach to Performance Modelling.* Cambridge Univ. Pr., New York (1996)

19. Hogg, R. V., Craig, A. T.: *Introduction to Mathematical Statistics.* 4th edn. Macmillan, New York (1978)
20. Ibe, O. C., Trivedi, K. S.: Stochastic Petri net models of polling systems. *IEEE J. on Sel. Areas in Comm.* **8**(9) (1990) 1649–1657
21. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. *Inf. and Comp.* **88**(1) (1990) 60–87
22. Karlin, S., McGregor, J. L.: The differential equations of birth-and-death processes, and the Stieltjes moment problem. *Trans. of the AMS* **85**(2) (1957) 489–546
23. Katoen, J.-P., Khattri, M., Zapreev, I. S.: A Markov reward model checker. In: *Quantitative Evaluation of Systems*, Los Alamitos, IEEE CS (2005) 243–244
24. Katoen, J.-P., Zapreev, I. S.: Safe on-the-fly steady-state detection for time-bounded reachability. In: *Quantitative Evaluation of Systems: QEST*, Los Alamitos, IEEE CS (2006) 301–310
25. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In Field, T., Harrison, P. G., Bradley, J., et al., eds.: *Computer Performance Evaluation: TOOLS. LNCS*, Vol. 2324, Berlin, Springer (2002) 200–204
26. Lassaigne, R., Peyronnet, S.: Approximate verification of probabilistic systems. In Hermanns, H., Segala, R., eds.: *Process Algebra and Probabilistic Methods: PAPM–PROBMIV. LNCS*, Vol. 2399, Berlin, Springer (2002) 213–214
27. Lecca, P., Priami, C.: Cell cycle control in eukaryotes: A BioSpi model. Technical Report DIT-03-045, Informatica e Telecommunicazioni: University of Trento (2003)
28. Mohanty, S. G., Montazer-Haghighi, A., Trueblood, R.: On the transient behavior of a finite birth-death process with an application. *Computers and Operations Research* **20**(3) (1993) 239–248
29. Norman, G., Shmatikov, V.: Analysis of probabilistic contract signing. *J. of Computer Security* **14**(6) (2006) 561–589
30. Oldenkamp, M.: Probabilistic model checking: A comparison of tools. MSc thesis, Univ. of Twente, Netherlands (2007) http://www.cs.utwente.nl/∼oldenkampha/.
31. Penna, G. D., Intrigila, B., Melatti, I., Tronci, E., Zilli, M. V.: Finite horizon analysis of Markov chains with the Murphi verifier. *Softw. Tools for Technology Transfer* **8**(4-5) (2006) 397–409
32. Pnueli, A., Zuck, L. D.: Verification of multiprocess probabilistic protocols. *Distributed Comput.* **1**(1) (1986) 53–72
33. PRISM: Probabilistic symbolic model checker. http://www.cs.bham.ac.uk/∼dxp/prism/ (2006)
34. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In Alur, R., Peled, D. A., eds.: *Computer Aided Verification: CAV. LNCS*, Vol. 3114, Berlin, Springer (2004) 202–215
35. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In Etessami, K., Rajamani, S. K., eds.: *Computer Aided Verification: CAV. LNCS*, Vol. 3576, Berlin, Springer (2005) 266–280
36. Shedler, G. S.: *Regenerative Stochastic Simulation.* Academic Pr., London (1993)
37. Somenzi, F.: CUDD: CU decision diagram package. http://vlsi.colorado.edu/∼fabio/CUDD/ (1997) Public software.
38. Younes, H. L. S.: Ymer: A statistical model checker. In Etessami, K., et al., eds.: *Computer Aided Verification. LNCS*, Vol. 3576, Berlin, Springer (2005) 429–433
39. Younes, H. L. S., Kwiatkowska, M., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. *Softw. Tools for Technology Transfer* **8**(3) (2006) 216–228
40. Younes, H. L. S., Simmons, R. G.: Probabilistic verification of discrete event systems using acceptance sampling. In Brinksma, E., Larsen, K. G., eds.: *Computer Aided Verification: CAV. LNCS*, Vol. 2404, Berlin, Springer (2002) 223–235