Distribution, Approximation and Probabilistic Model Checking

Guillaume Guirado

EPITA Research and Development Laboratory (LRDE)

Thomas Herault

LRI, University Paris South-XI

Richard Lassaigne

Equipe de Logique Mathématique, UMR 7056 CNRS, University Paris VII

Sylvain Peyronnet

EPITA Research and Development Laboratory (LRDE)
syp@lrde.epita.fr

Abstract

APMC is a model checker dedicated to the quantitative verification of fully probabilistic systems against LTL formulas. Using a Monte-Carlo method in order to efficiently approximate the verification of probabilistic specifications, it could be used naturally in a distributed framework. We present here the tool and his distribution scheme, together with extensive performance evaluation, showing the scalability of the method, even on clusters containing 500+ heterogeneous workstations.

1 Introduction

Probabilistic model checking is an algorithmic method that aims to automatically verify that quantitative properties holds in probabilistic systems. The main drawback of the method is the so-called state space explosion phenomenon, that is the fact that workstations run out of memory while verifying large probabilistic systems. A usual direction of research to address this problem is to design distributed model checking algorithms in order to handle larger systems. Most of these methods are about the distribution of the state space on several machines.

In the last couple of years, we showed that a completely different approach can be used in order to save space while verifying large systems. Indeed, we proposed to use approximate probabilistic model checking. The idea of using approximation becomes more and more popular and is now used by several research groups [16,4]. Our approach [5] is more precisely based on the sampling of execution paths of the probabilistic system. This method is, by construction, massively parallel. Indeed, one can distribute the computation on a large cluster of machines in the following way: each machine generates execution paths and verifies the specification on each of these paths, then sends the obtained results to a master. After a certain time of computation, the master received enough results to conclude on the (approximate) validity of the specification on the system.

In this paper, we explain in details the method we developed and we analyze the performances of our methodology on very large clusters of heterogeneous machines (up to 500 machines). All the experiments were done using APMC (Approximate probabilistic Model Checker), which is the tool that implements our method.

The paper starts with a short review of the related work. Then, in section 3, we give the theoretical foundations of our tool APMC and explain his architecture and implementation. Last, we present in section 4 the results of extensive experiments on various case studies and sets of machines. These experiments show the scalability of the approach.

2 Related Work

In the last few years, distributed model checking have gained a renewal of interest, due to the emergence of easily available "computing farms", that is very large set of machines usable for computation. There is now a challenge of using such clusters in every domain of computer science. Several methods have been developed in order to speed up the model checking and/or avoid the state space explosion phenomenon.

One of the first idea in the use of parallelization was to distribute the construction of the state-space (see for example [13,3]). Basically this is done using a partition of the set of the reachable states by way of a hashing function, this partition induces a natural parallelization.

Concerning the manipulation of the state space, an other way of research is to improve the size of the transitions systems that can be handled by the model checker. Out of core method were designed to do this [10,9], particularly for probabilistic systems (that is Markov models) .

A lot of others methods have been developed and discussed [7], but in the rest of the papers, we won't discuss anymore about all these methods, since none of them distribute the whole process of the verification in a massively distributed way (e.g. hundreds of machines).

The method we designed for the distributed and approximate verification of probabilistic systems is completely different since it is naturally a parallel method (due to the use of a Monte-Carlo sampling technique). There already exists other sampling techniques for the verification of probabilistic systems [16,4]. The method of [16] uses the framework of hypothesis testing while [4] uses also a monte-Carlo method. These two methods have also the potential of being parallelized, but, to our knowledge, it wasn't done by now.

3 Approximate Probabilistic Model Checking

3.1 Theoretical Foundations

The APMC approach [5] uses an efficient Monte-Carlo method to approximate satisfaction probabilities of monotone properties over fully probabilistic transitions systems. Properties to be checked are expressed in LTL: Linear Temporal Logic.

3.1.1 APMC method

LTL formulas are built over a set of atomic propositions labeling states.

Definition 3.1 A fully probabilistic transition system (PTS or DTMC) is a tuple $\mathcal{M} = (S, \overline{s}, P)$ where S is a set of states, $\overline{s}$ is the initial state, and P is a transition probability function.

We denote by $Path(s)$ the set of paths whose first state is s. The length of a path π is the number of states in the path and is denoted by $|\pi|$, this length can be infinite. The probability measure $Prob$ over the set $Path(s)$ is defined in a classical way [8]. We denote by $Prob[\phi]$ the measure of the set of paths $\{\pi \mid \pi(0) = s \text{ and } \mathcal{M}, \pi \models \phi\}$ (see [15]). Let $Path_k(s)$ be the set of all paths of length $k > 0$ starting at s in a PTS. The probability of an LTL formula ϕ on $Path_k(s)$ is the measure of paths satisfying ϕ in $Path_k(s)$ and is denoted by $Prob_k[\phi]$.

Definition 3.2 An LTL formula ϕ is *monotone* if and only if for all $k > 0$, for all paths π of length k, $\mathcal{M}, \pi \models \phi \implies \mathcal{M}, \pi^+ \models \phi$, where π^+ is any path of which π is a prefix.

A basic property of monotone formulas is the following one: if ϕ is a monotone formula, $0 < b \leq 1$ and if there exists some $k \in \mathbb{N}^*$ such that $Prob_k[\phi] \geq b$, then $Prob[\phi] \geq b$.

In order to verify some probabilistic specification $Prob[\phi] \geq b$, we choose a first value of $k = O(log|S|)$, then we approximate the probability $Prob_k[\phi]$ and test if the result is greater than b. If $Prob_k[\phi] \geq b$ is true, then the monotonicity of the property guarantees that $Prob[\phi] \geq b$ is true. Otherwise, we increment the value of k and approximate again $Prob_k[\phi]$. We iterate this procedure within a certain bound which, in many cases, is logarithmic in the number of states. In the worst case, this bound is strongly related to the rapid mixing rate of the underlying Markov chain [12]. If the results of all tests $Prob_k[\psi] \geq b$ are negative, then we can conclude that $Prob[\psi] \ngeq b$. If we are interested only with probabilistic time bounded properties, as here, we can set k to the maximum time bound in subformulas of the specification. In the following, we describe how to approximate efficiently the probability $Prob_k[\phi]$.

3.1.2 Randomized approximation scheme

In order to estimate the probabilities of monotone properties with a simple randomized algorithm, we generate random paths in the probabilistic space underlying the DTMC structure of depth k and compute a random variable A/N which estimates $Prob_k[\psi]$. To verify a statement $Prob_k[\psi] \geq b$, we test whether $A/N > b - \varepsilon$. Our decision is correct with confidence $(1 - \delta)$ after a number of samples polynomial in $\frac{1}{\varepsilon}$ and $\log \frac{1}{\delta}$. The main advantage of the method is that we can proceed with just a succinct representation of the transition graph, that is a succinct description in an input language, which is the same in PRISM [1]. Our approximation problem is defined by giving as input x a succinct representation of a MDP, a formula and a positive integer k. The succinct representation is used to generate a set of execution paths of length k. A randomized approximation scheme is a randomized algorithm which computes with high confidence a good approximation of the probability measure $\mu(x)$ of the formula ϕ over the set of execution paths.

Definition 3.3 A fully polynomial randomized approximation scheme (FPRAS) for a probability problem is a randomized algorithm $\mathcal{A}$ that takes an input x, two real numbers $0 < \varepsilon, \delta < 1$ and produces a value $A(x, \varepsilon, \delta)$ such that:

$$Prob\big[|A(x, \varepsilon, \delta) - \mu(x)| \leq \varepsilon\big] \;\geq\; 1 - \delta.$$

The running time of $\mathcal{A}$ is polynomial in $|x|$, $\frac{1}{\varepsilon}$ and $\log \frac{1}{\delta}$.

The probability is taken over the random choices of the algorithm. We call ε the *approximation parameter* and δ the *confidence parameter*. The APMC approximation algorithm consists in generating $O(\frac{1}{\varepsilon^2}.\log \frac{1}{\delta})$ paths, verifying the formula ϕ on each path and computing the fraction of satisfying paths.

Theorem 3.4 *The APMC approximation algorithm is a fully randomized approximation scheme for the probability $p = Prob_k[\psi]$ of an LTL formula ψ if $p \in]0, 1[$.*

This result is obtained by using Chernoff-Hoeffding bounds [6] on the tail of the distribution of a sum of independent random variables. The complexity of the algorithm depends on $\log(1/\delta)$, this allows us to set δ to very small values. The dependence in ε is much more crucial, since the complexity is quadratic in $1/\varepsilon$.

3.2 Architecture of APMC

APMC architecture is twofold, as described in figure 1. The first component, the APMC Compiler produces an had doc verifier including a sample generator and a checker for a given model (described in Reactive Modules) and a given

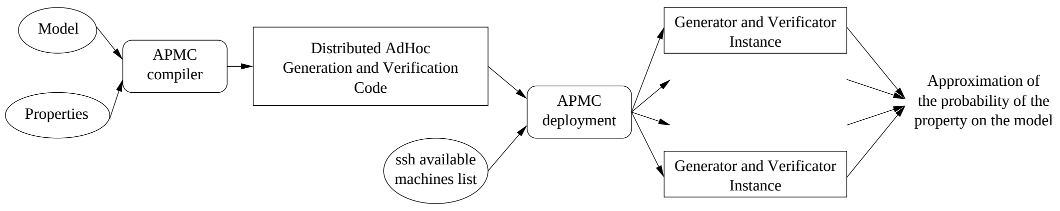

Model
Properties
APMC compiler
Distributed AdHoc Generation and Verification Code
ssh available machines list
APMC deployment
Generator and Verificator Instance
Generator and Verificator Instance
Approximation of the probability of the property on the model

Fig. 1. APMC components

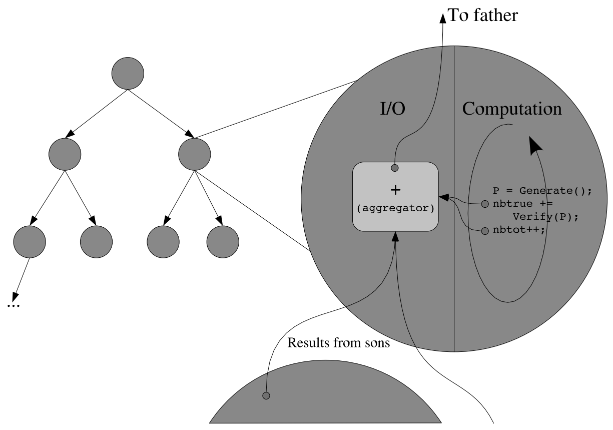


Fig. 2. APMC deployment scheme

property (LTL). The second module, the APMC Deployer, takes this verifier and the set of available computing resources, deploys the verifier on this set of computers and collects the result, which is the approximated value of the probability of the initial formula on the model.

The technique used to approximate this value assumes the verification of the formula on a large set of independent samples of bounded length. We use the independence property of the samples to parallelize the generation and verification of each sample.

The deployment is performed following a spanning tree of bounded arity. Each node of the tree runs on a single computing resource, and spawns children up to the bound on other available resources. While its father still accepts results from it, and until the number of collected samples is greater than the requested number if it is the root, it generates a sample and verifies the property on it. At each verification, the counters of false and true samples are updated. Regularly (that is on a fixed timeout), each node sends its counters of false and true samples to its father, and resets them (except for the root, which awaits the end of the computation to produce these numbers). When a node receives these counters from one of its children, it aggregates these numbers as if it produced the verification (see figure 2).

This deployment technique is assumed to be scalable, since the number and amount of data of all communications with a same node depends only on the arity bound of the tree. The tree topology was chosen to reduce the starting time, which is proportional to the depth of the tree, hence logarithmic in the number of computing resources. It also provides a logarithmic latency to aggregate the results from all nodes in the root. A drawback of this method is that the system may over generate and verify some samples (which does not precludes the validity of the final result, but may provide a better approximation than requested), up until the root claims that enough samples have been generated, and the tree is destroyed. This diffusion is also linear in the height of the tree and proportional to the communication timeout.

As for the parallelization, the technique provides a simple solution for fault tolerance: since each generation and verification is independent from the others, some of these verifications may be lost without consequences on the quality of the result. Thus, if a computing node crashes, its children will presume that the computation is finished and will stop running; its father detects it and simply spawns a new subtree. All the workers of the subtree rooted at the crashed process are assumed lost and free to use again.

3.3 Implementation

The APMC software consists of three independent components: the parser, the core library and the deployment tool. This design provides the possibility to include the engine (core library) in many model checkers, like we are doing with the PRISM tool [11].

The parser is a simple lex/yacc program which parses a sub-language of the PRISM language (Reactive Modules [1]), and a simple language for LTL formulas. It then calls the core APMC library to produce an internal succinct representation of the model (linear in the size of the Reactive Modules file), and of the properties (linear in the size of the property file).

The library then produces the ad-hoc generator and verifier as an ANSI C code (the generator/verifier is a standalone program deployed by the Deployer). APMC implements three strategies to generate the code of this program with respect to the synchronizations of the Reactive Modules: the first one (called *sync at compile-time*) pre-computes all the combinations of rules, thus building the synchronized succinct model representation, where each rule is not synchronized. This is the most efficient strategy with respect to time, but it is the most memory consuming strategy. At runtime, the generator simply evaluates each guard on the current configuration, building the set of fireable rules. A rule is chosen randomly between these fireable rules and the action is triggered to compute the next configuration. The second strategy (*sync at run-time*) is provided to seize larger, highly synchronized, models. There, the evaluation of the guards is done together with the computation of the synchronizations, which is thus done at each simulation step, spending more time to compute the set of fireable rules, but using much less memory. This strategy is used only when the model induces a lot of synchronization and the generated code prohibits efficient compilation. The last strategy is an improvement of the first one: when most the time, the number of fireable rules is high, instead of first computing the fireable rules (thus evaluating each guard on the configuration), a rule is chosen uniformly, and if its guard is true, its action is triggered. If its guard is false, another rule is chosen randomly.

The main loop of the code produced by the library consists in generating a path (i.e. a set of configurations) of given length, and evaluating the property (temporal path formula) on each path. The number of iterations of this loop is a parameter to the program.

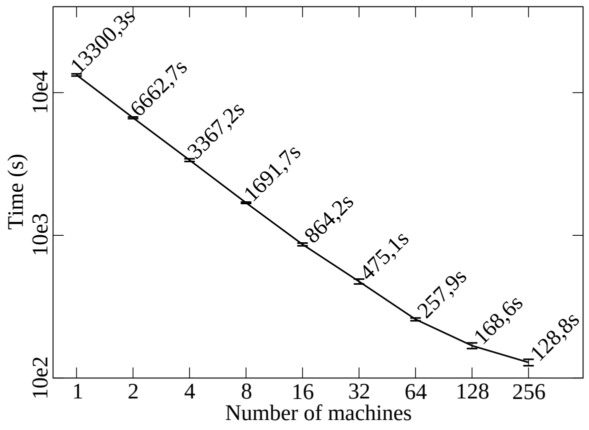
The last component of the APMC software is the deployment tool. This tool takes the code produced by the library, compiles it on different architectures and deploys the programs on a set of computing nodes following a regular spanning tree of bounded arity. The program executed on the nodes includes two parts: an I/O part, and a computing part. The computing part is generated by the core library, while the I/O part is generic. This I/O part implements the spanning tree. It handles the connexions with the children and with the father of the node. Father connexion is handled through the standard output. Messages are sent regularly to the father, according to the algorithm described in the architecture section. When this file descriptor is closed, the computation is stopped and the program exits. Children connexions are handled using a double pipe with an ssh (or rsh) command. The deployment tool comes with a set of shell scripts passed to the ssh command. These scripts download and compiles for the new spawned computing node the generated code, split the list of available resources between the children and launches recursively the compiled program on the node. This technique does not presume the existence of NFS, or other file sharing system. Currently, we assume that each node provides a remote shell service (ssh or rsh), and the autotools, Make and a C compiler. Current work in progress will assume only the C compiler and will reduce the amount of needed compilations by factorizing the compilations for each kind of architecture, instead of doing a compilation on each machine.

4 Performance Evaluation

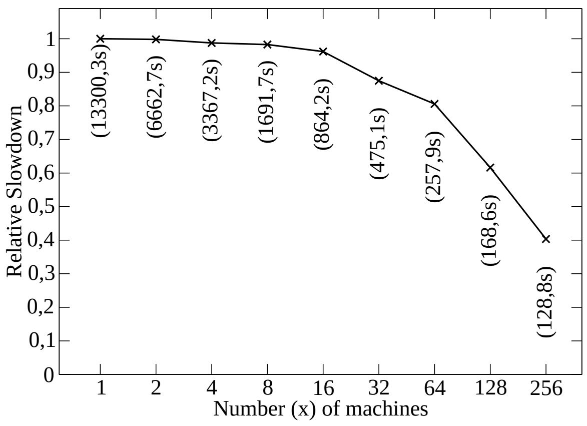
The experimental platform consists in 500 Athlon 3000+ with 1Gb of RAM, running under NetBSD 1.6.1, 100Mb ethernet network. The remote shell program used is OpenSSH, with public key authentication. The compiler on each worker is gcc-2.95.3, with the -03 option.

All the measurement are done on the dining philosopher problem, checking a double accessibility property. The dining philosopher problem [14], being well studied, allows us to separate between the phenomenons due to the tool and those due to the model itself. Since this model does not include synchronizations, we conducted all experiments using the most efficient strategy, “sync at compile-time”.

The first set of figures (figure 3) describes the acceleration in time due to



(a) Time against the number of workers



(b) Relative slowdown

Fig. 3. Time of model checking for the randomized dining philosophers problem

the parallelization. To obtain these results, we ran APMC on the 160 dining philosopher problem [14], on an increasing number of workers following the binary tree deployment described in section 3.2. For all these experiments, we set $\epsilon = 10^{-2}, \delta = 10^{-10}$ (that is a generation of 940,000 paths by experiment), which are usual values for these parameters, and $k = 200$. On the curves are represented the mean value of a set of 80 measures by point.

The first figure 3(a) shows the total execution time as function of the number of workers on a double logarithmic scale. One can see that, as expected, the execution time decreases quickly as the number of workers increases. The figure also shows a slowdown in the linear acceleration when having more than 64 workers.

The next figure 3(b) focuses on this phenomenon. On the x axis, is the number of workers, and on the y axis is the relative slowdown given by the formula $y_x = t_1/(x \times t_x)$ where t_x is the time measured in figure 3(a) for the given x. With this measure, the value 1.0 represents a perfect scalability, whereas smaller values demonstrate a lower use of the whole system.

One can see that when using more than 32 workers, the relative slowdown is higher than 10% on this example. The deployment phase is time consuming, and starting at 32 workers, the deployment duration is not negligible compared to the computation time. This accumulated time consumption is exponential in the depth of the tree (that is linear in the number of workers), nonetheless each worker waits at most for a logarithmic time before beginning its execution, which explains why adding workers is an improvement up to the amount where the computation ends before launching the last workers.

Figure 4 shows the time needed to verify the model with the same parameters as in figure 3 on a cluster of 256 workers, as function of the arity of the deployment tree. Obviously, except the case of arity 1 (a string of workers), increasing the arity of the tree does not improve significantly the performances of the deployment system. On the other hand, increasing the

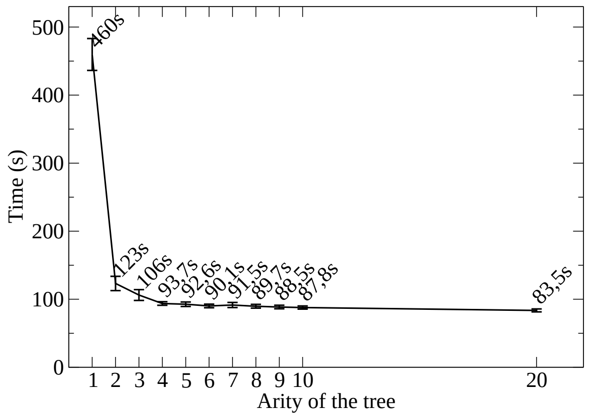


Fig. 4. Time of model checking according to the arity of the deployment tree

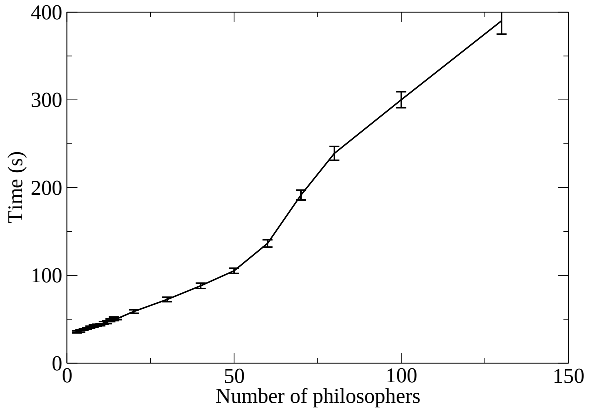


Fig. 5. Time of model checking against the number of philosophers

arity does not hinder the performances, so as the figure 3 teaches us, when the relative slowdown becomes too large, it makes sense to increase the arity in order to decrease the depth of the tree.

The last figure 5 presents the time needed to verify the 3 to 130 dining philosopher problems. All verifications were done on paths of length 200. 32 workers were used to verify 940,000 paths. The aim of this experiment is to evaluate the generation of the code. Indeed, since all verifications use the same path length and the same number of paths, the time differences are only due to the quality of the generated code.

One can see that the curve is in three linear parts. The main loop of the code consists in iterating over all the guards of the model (which are functions of the program). The number of guards increases linearly with the number of philosophers. So it is natural that the time needed to iterate over all the guards is linear in the number of philosophers. It is less expected that the curve includes three different slopes. Since the generated code occupies up to 256Kb more resident memory with the 130 philosophers problem than with the 3 philosopher problem, we suspect that this is due to CPU code cache invalidations.

As a validation of APMC as a cycle stealing verification tool, we conducted another experiment including all the available computers of the EPITA school of computer science. We verified the 160 dining philosophers problem on a platform of 500 computers used by other applications at the same time. We conducted two experiments: the first where 940,000 paths were generated, the other one with 9,400,000. The first experiment took 99 seconds, the second one 446 seconds.

It is interesting to note that, although the amount of computation needed in the second experiment was ten times higher than for the first, the time needed to complete it was only 4.5 times higher. It is due to the fact that for the first experiment, the system does not have enough time to take advantage of the full platform.

5 Discussion

Traditionally, model checking is a highly expensive computational activity. The main drawback of the method is the memory needed to finalize the verification of large systems. "Classical" distributed model checking aims to lower the memory cost by distributing the state space. Using approximation techniques, we can trade the memory cost with simple computations on a large number of system executions paths. This is the point where we can massively distribute the process, by partitioning the sample into sets that are independently processed.

Using this method, we can verify very large systems using a constant amount of memory (when k is fixed). The power of computation usable for the verification is limited only by the number of available computers.

However, experiments show that for each systems, there is no gain when having more than a critical number of machines. This is due to the fact that the deployment scheme has a cost.

APMC is also interesting from an economic point of view. Since APMC runs in background using very few memory, it can run on classical desktop machines (implementing cycle stealing techniques), thus avoiding the cost of an expensive cluster of dedicated workstations.

References

[1] R. Alur and T. Henzinger. Reactive modules. in Proc. of *LICS 96*, 1996.

[2] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Formal Methods in*

Computer-Aided Design, Third International Conference, (FMCAD'00), volume 1954 of LNCS, pp 390–404, 2000.

[3] H. Garavel, R. Mateescu and I. Smarandache. Parallel state space construction for model-checking. Proceedings of the 8th International SPIN Workshop on Model Checking of Software SPIN'2001 (Toronto, Canada), volume 2057 of LNCS, pages 217–234, 2001.

[4] R. Grosu and S. A. Smolka. Monte Carlo Model Checking. In the proc. of *TACAS 2005.* pp271–286, LNCS 3440. 2005.

[5] T. Herault, R. Lassaigne, F. Magniette and S. Peyronnet. Approximate Probabilistic Model Checking. In *Proceedings of Fifth International VMCAI'04*, pp 73-84, LNCS:2937, January 2004.

[6] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13-30, 1963.

[7] C. P. Inggs and H. Barringer. On the parallelisation of model checking. In *Proceedings of the 2nd Workshop on Automated Verification of Critical Systems (AVOCS'02).* Technical report, University of Birmingham, April 2002.

[8] J. Kemeny, J. Snell and A. Knapp. *Denumerable markov chains.* Springer-Verlag, 1976.

[9] W. J. Knottenbelt and P. G. Harrison. Distributed disk-based solution techniques for large Markov models. In Proc. of NSMC'99, 3rd International Workshop on the Numerical Solution of Markov Chains, September 1999.

[10] M. Kwiatkowska, R. Mehmood, G. Norman and D. Parker. Symbolic Out-of-Core Solution Method for Markov Model. In Proc. Workshop on Parallel and Distributed Model Checking (PDMC'02), volume 68.4 of ENTCS, August 2002

[11] M. Kwiatkowska, G. Norman and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, pp 200-204, LNCS:2324, 2002.

[12] L. Lovasz and P. Winkler. Exact mixing time in an unknown markov chain. *Electronic journal of combinatorics*, 1995.

[13] D. M. Nicol and G. Ciardo. Automated Parallelization of Discrete State-Space Generation. *J. Parallel Distrib. Comput.*, 47(2):153-167, 1997.

[14] A. Pnueli and L. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, pages 1:53–72, 1986.

[15] M.Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. *Proc. 26th Annual Symposium on Foundations of Computer Science*, 1985.

[16] H. L. S. Younes and R. G. Simmons. Probabilistic Verication of Discrete Event Systems using Acceptance Sampling. *In Proc. of the 14th International Conference on Computer Aided Verification*, LNCS, 2404:223–235. 2002.