A Case Study on Exploring the Performance Limits of PRISM[*]

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at

Tamás Bérczes and János Sztrik
Department of Informatics Systems and Networks, Faculty of Informatics
University of Debrecen, Debrecen, Hungary
berczes.tamas@inf.unideb.hu, sztrik.janos@inf.unideb.hu

Gábor Kusper
Eszterházy Károly College, Eger, Hungary
gkusper@aries.ektf.hu

September 23, 2013

Abstract

We investigate based on a simple system model the range of applicability of the probabilistic model checker PRISM. In particular, we analyze the effect of the choice of the checking engine, of the numerical equation solver, and of other settings on the time and memory required for the analysis of the model by PRISM. The results demonstrate that not always the same engine is the fastest one but that for large models an otherwise slow engine may show superior performance; furthermore, the appropriate choice of the termination criterium can make the crucial difference between the success and the failure of an analysis.

[*]Supported by the project HU 10/2012 of the Austrian Academic Exchange Service (ÖAD) and by the TÁMOP-4.2.2.C-11/1/KONV-2012-0001 project (this project has been supported by the European Union, co-financed by the European Social Fund).

1. Introduction

Evaluating the performance of probabilistic model checking is a tricky task. The theoretical complexity of CSL model checking is known to be linear in the size of the formula and polynomial in the size of the state space (Section 4.6 of [3]). It also entails the solution of a linear equation system whose size is the number of states; this solution can be derived in cubic time. However, this knowledge does not give any good estimation for the concrete size of a system that can be practically checked in a reasonable amount of time with current implementations and computer technology.

In this paper we evaluate by a small case study the limits of the practical applicability of the probabilistic model checker PRISM [2, 4] which we have already evaluated with respect to its capability for timing analysis [5, 6] and used for the performance analysis of various computing models such as mobile cellular networks [7, 9, 8]. In particular, we investigate the effect of the choice of the checker engine and of the equation solver to be applied for the analysis (PRISM supports various engines and solvers in numerous combinations).

The case study is based on the PRISM model listed in Appendix A which is derived from a model originally developed by Tamas Berczes and subsequently adapted by Wolfgang Schreiner: it consists of S servers to which jobs are distributed by N clients (the listing shows the case $S = 24$, the other cases can be derived by simple adaptations of the model). Every server has a local job queue of size J from which it takes the next job and processes it at rate μ. Every client generates a job at rate λ and sends it to that server that currently has the smallest number of jobs in its queue; thus a "fair" distribution of load among the servers is established. The model is analyzed for the average load of one server by the query

```
"mS1" : R{"mS1"}=? [ S ] ;
```

Combinatorics tells us that the state space size of this model is bounded by $q(N,S+1)\cdot(S+1)!$ where $q(n,k)$ is the number of partitions of n into at most k classes [1]. The partition function q is approximated for $n \to \infty$ with fixed k as

$$q(n,k) \approx \frac{n^{k-1}}{k!(k-1)!}$$

Thus the state space size of our model is bounded by $N^S/S!$ which is polynomial in N and exponential in S. However, these asymptotic complexity results tell us little about the actual size of the state space and of the actual memory consumption and model checking times for concrete values of N and S; this is what we are going to evaluate in the following section.

2. The Analysis

In our analysis, we have applied the latest version 4.1beta2 of PRISM to our model with $S \in \{8,16,24\}$ and $N \in \{10,15,20,25\}$; since $25/8 < 4$ and the clients fill up the load of servers "evenly", it suffices to take $J = 4$ without encountering the situation that a server with full job queue is chosen as the recipient of a new job (actually, we used $J = 5$). The experiments were

$S = 8, \varepsilon = 0.050$	10	15	20	25
sparse/gs				
sparse/jor				
hybrid/gs				
hybrid/jor				
mtbdd/jacobi				
mtbdd/jor				

$S = 16, \varepsilon = 0.050$	10	15	20	25
sparse/gs				
sparse/jor				
hybrid/gs				
hybrid/jor				
mtbdd/jacobi	×			
mtbdd/jor				

$S = 24, \varepsilon = 0.050$	10	15	20	25
sparse/gs				×
sparse/jor				
hybrid/gs				×
hybrid/jor				
mtbdd/jacobi	×	×		
mtbdd/jor				

$S = 8, \varepsilon = 0.010$	10	15	20	25
sparse/gs				
sparse/jor				
hybrid/gs				
hybrid/jor				
mtbdd/jacobi				
mtbdd/jor				

$S = 16, \varepsilon = 0.010$	10	15	20	25
sparse/gs				
sparse/jor				
hybrid/gs				
hybrid/jor				×
mtbdd/jacobi	×			×
mtbdd/jor				

$S = 24, \varepsilon = 0.010$	10	15	20	25
sparse/gs				×
sparse/jor				×
hybrid/gs				×
hybrid/jor				×
mtbdd/jacobi	×	×		
mtbdd/jor				

$S = 8, \varepsilon = 0.005$	10	15	20	25
sparse/gs				
sparse/jor				
hybrid/gs				
hybrid/jor				
mtbdd/jacobi				
mtbdd/jor				

$S = 16, \varepsilon = 0.005$	10	15	20	25
sparse/gs				
sparse/jor				
hybrid/gs				
hybrid/jor				
mtbdd/jacobi	×			×
mtbdd/jor				×

$S = 24, \varepsilon = 0.005$	10	15	20	25
sparse/gs				×
sparse/jor				×
hybrid/gs				×
hybrid/jor				×
mtbdd/jacobi	×	×		
mtbdd/jor				

Figure 1: Analysis Failures

mostly performed on a laptop with an Intel Core i7-2670QM CPU running at 2.2GHz and 16GB of memory (except for some large-scale experiments which were performed on a server with Intel Xeon E78837 CPUs running at 2.67 GHz and several TB of memory; since this type of processor is about 40% faster, we have inflated the timings derived on the server by 40% to make them comparable with the figures derived on the laptop).

Checker Engines and Equation Solvers In our experiments, we have applied the following combinations of checker engines and equation solvers supported by PRISM:

- The "sparse" engine is based on a matrix representation of the system which requires most space but is supposed to be fastest. We use with this engine the "Gauss-Seidel" (GS) solver (which we find generally faster than the default "Jacobi" solver) and as an alternative the "JOR" (Jacobi Over-Relaxation) solver.
- The "MTBDD" engine uses a representation of the system based on Multi-Terminal Binary Decision Diagrams (MTBDDs); this representation is the most memory-efficient one but is also supposed to be slowest; since the engine does not support the GS solver, we apply the "Jacobi" and the JOR solver.
- The "hybrid" engine uses a combination of matrices and MTBDDs which is much more memory-efficient than the "sparse" engine but still supposed to be faster than the MTBBD engine. We use for this engine, like for the "sparse" one, the GS and the JOR solver.

The most recent version of PRISM also supports a new "explicit" engine which is based on a completely explicit state representation of the system (unlike the three engines above which use a representation that is at least partially symbolic). This engine is advised only for systems where the size of the state space is small or where only a small number of states is reachable. Since we are interested in big models with many reachable states, we have not considered it for our analysis.

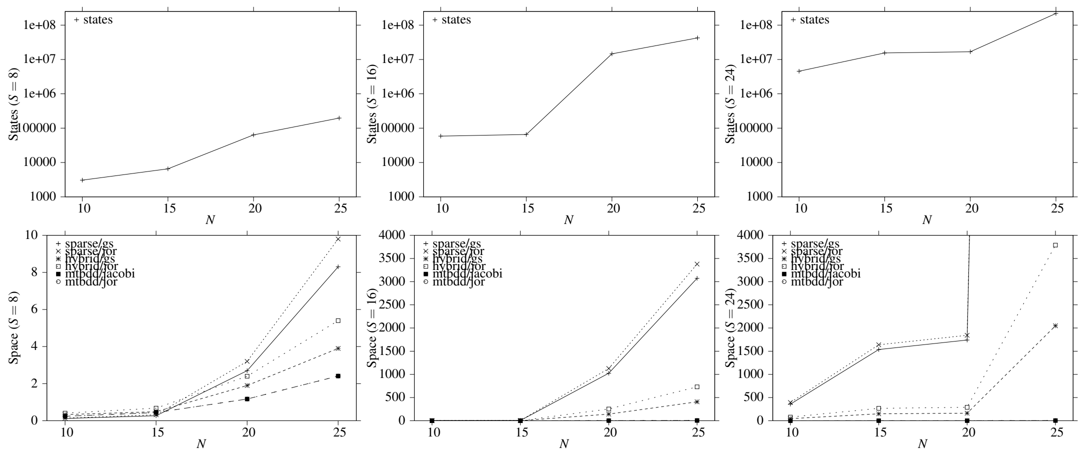

Figure 2: State Space Size and Memory Consumption (MB)

However, as the following presentation will show, not every combination of checking engine and equation solver did actually produce a result for every analysis, i.e., the equation solver did in some cases not converge to a solution (in several hours). For easier reference, Figure 1 illustrates all those combinations of model parameters where the analysis failed; each table refers to one combination of number of servers S and termination criterion ε (explained below), each column in a table refers to a particular value of the number of clients N.

State Space Sizes and Memory Requirements The top row in Figure 2 illustrates the state space sizes of the systems for the various values of S and N; for $S = 24$ and $N = 20$ we get about 16 million states, for $N = 25$ about 218 million states. The bottom row illustrates for these systems the memory consumption (in MB) reported by PRISM for the allocation of its data structures.

Our experiments essentially confirm the expectations with respect to memory efficiency; "MTBDD" reports generally very moderate space requirements, "hybrid" can cope with 4GB memory in all cases; "sparse" has highest requirements but also copes with 4GB except for the case of $S = 24$ and $N = 25$ where the requirements explode to more than 80 GB; such checks can only be performed on appropriately equipped server machines as the one we used for the corresponding experiments.

However, the diagrams only show half of the truth: the figures reported by PRISM only refer to the size of the allocated data structures which seems to be in general lower than the actual memory requirements of the PRISM process (as displayed by the operating system for the resident memory size of the process). Both figures are reasonably close for the "sparse" and the "hybrid" engine but they vastly differ for the "MTBDD" engine: here the process consumes for $S = 24$ and $N = 25$ several GB of memory (similar to the "hybrid" engine) while PRISM reports only 4MB of memory used.

Thus the hybrid and the "MTBDD" engine cope in all cases with a reasonably limited amount of memory, although the requirements for MTBDD are (taking the memory consumed by the

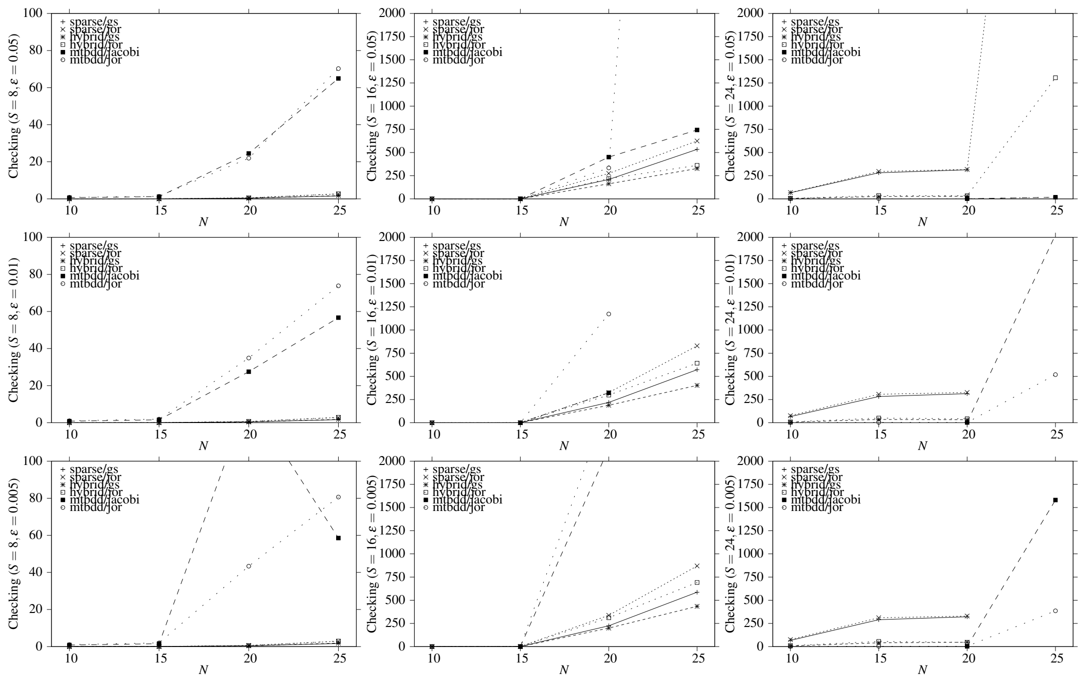

Figure 3: Model Checking Times (in s)

process as the criterion) not so much smaller as one might think. On the other side, the "sparse" engine copes only up to a few million of states with a reasonable amount of memory; if the size of the state space reaches the order of 100 millions, the requirements explode.

PRISM with 64 bit Indices When executing the experiment with the "sparse" engine and large state spaces, we encountered the curious problem that PRISM reported "Out of memory" even if the underlying server machine provided sufficiently much main memory. Since PRISM is open source, we dug deeper into the problem and detected the core reason: the "sparse" engine allocates arrays whose lengths and indices are denoted by the C++ datatype `(unsigned) int` which is by default 32 bit; if the number of requested entries gets bigger than the maximum value (2^{31} respectively 2^{32}) of this type, this number overflows and becomes negative; consequently the memory allocation operation fails.

We have reported this fact to Dave Parker, the core developer of PRISM, and patched the code for our experiments to use 64bit integers (C++ datatype `(unsigned) long`) for array lengths and indices; with this patched version, we were able to successfully perform the corresponding experiments.

Checking Times Figure 3 reports the model checking times for the various model parameters and engine/checker combinations. We used the value $\varepsilon \in \{0.05, 0.01, 0.005\}$ as the relative error for detecting the convergence of the solution of the numerical equation system, because we experienced with the default $\varepsilon = 10^{-6}$ divergence in many cases; a relative error of 5%, 1%, and 0.1% for the computed value seems acceptable in practice.

For $S = 8$, all combinations of engines/solvers yield a result in an acceptable amount of time; however, while the "sparse" and the "hybrid" engine terminate after a few seconds, the "MTBDD" engine already needs up to 80 seconds; for $N = 16$ and $\varepsilon = 0.005$, the Jacobi solver applied in the "MTBDD" engine takes more than 130s, due to slow convergence of the solver. This is the first instance of the fact that the convergence criterion may have a big impact on the convergence of the "MTBDD" engine, with both the Jacobi and the JOR solver.

For $S = 16$ and $N \leq 20$, the solvers usually behaved quite decently, with the "sparse" engine being the fastest one, followed by the "hybrid" engine, and already with some distance by the "MTBDD" engine. For $\varepsilon \leq 0.01$, the "MTBDD" engine started to show slow convergence; for $S = 16$ and $N = 10$ the Jacobi solver did not converge at all in a reasonable amount of time. For $\varepsilon \leq 0.01$ and $N = 25$, the "MTBDD" engine did not show convergence any more. So all in all, the performance of MTBDD here disappointed.

However, $S = 24$ showed a crucially different picture, especially for $N = 25$:

- The GS solver did not converge at all in a reasonable amount of time, neither for the "sparse" not for the "hybrid" engine.

- Both the "sparse" and the "hybrid" engine behaved poorly and only yielded a result for $\varepsilon = 0.05$: the "hybrid" engine converged after about 1300s, while the "sparse" engine took more than 8300s.

  For the "sparse" engine, the analysis of the PRISM output revealed the interesting fact, that to the total execution time the solution of the equation system only contributed a minor fraction (about 1100s) while the vast majority (4700s) was spent in setting up the sparse matrix and associated vectors. For $\varepsilon = 0.01$, however, the solution of the equation system did not even converge after more than 20000s: each iteration of the solver started to take about 200s, so only 100 iterations and a relative accuracy of $\varepsilon = 0.035$ could be achieved within this time.

  Thus for the "sparse" engine and large state spaces, not only the available memory but also the solution time becomes a limiting factor, both for the setup of the equation system and for its actual solution. However, while the "hybrid" engine does not suffer from the long setup time, its overall results are also not much more attractive.

- Surprisingly, the "MTBDD" engine fared here best, with both the Jacobi solver and the JOR solver deriving solutions in a reasonable amount of time. However, the JOR solver was about 3 times faster than the Jacobi solver (which, also did not converge for $N \in \{15, 20\}$) and is thus preferable.

Thus, while for smaller state spaces the "sparse" engine with the Gauss-Seidel solver performed best, for large state spaces the combination of "MTBDD" engine with the JOR solver proved preferable, not only from the point of view of memory consumption, but (somewhat surprisingly) also from the point of view of computation time.

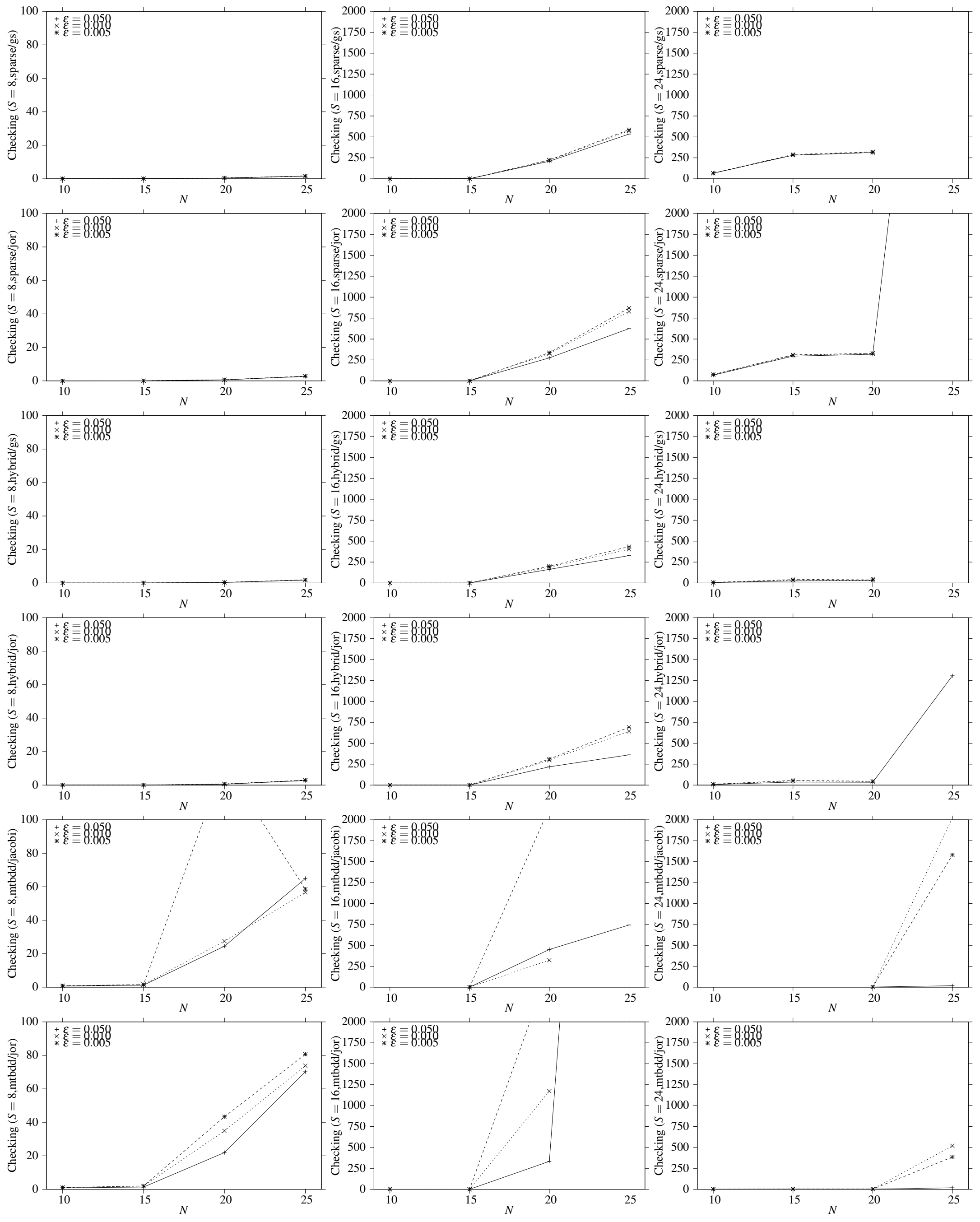

Figure 4: Model Checking Times (in s)

Effect of Solution Accuracy Figure 4 displays the results already illustrated in Figure 3, but organized in a different way, with one row for each solver. Thus we can depict the overall effect of the termination criterion ε on the checking time. We clearly see that it has limited effect on the "sparse" and on the "hybrid" engine (with the exception of sparse/JOR for $S = 24$ and $N = 25$ where convergence could be only achieved with $\varepsilon = 0.005$). However, it plays in many an cases an important role for the "MTBDD" engine, whose convergence may suffer drastically from a smaller ε, as can be seen in the diagrams for $S = 16$ and $S = 24$ where values of $\varepsilon \leq 0.01$ lead to much worse results.

3. Conclusions

As our small case study demonstrates, the range of applicability of PRISM is not easy to establish; the appropriate choice of the checking engine and of the numerical equation solver proves essential, but also the termination criterion turns out to play an important role. In a nutshell, our results are as follows:

- The combination of the "sparse" engine with the Gauss-Seidel solver showed consistently the best results for small to medium-size examples. For really large state spaces, it behaved however poorly; not only that it required a large amount of memory (which could be provided by large server machines), it also took an extremely long time to set up its data structures, and finally also the equation solver became painfully slow.

- For really large examples the combination of the "MTBDD" engine with the JOR solver proved (somewhat surprisingly) superior. However, the convergence rate was generally not as consistent as for the "sparse" engine; even for smaller examples, it did not converge several times.

- The "hybrid" engine did not really excel in any case; it just seems a compromise if one cannot make his/her mind up with respect to the choice of the checking engine. The use of the "hybrid" engine was mostly okay, except for really large state spaces, where it mostly suffered from the drawbacks of the "sparse" engine.

- The appropriate choice of the termination criterion should not be overlooked; by default, PRISM terminates a check only after a very high accuracy of the result could be established, but this may yield prohibitive computation times. If a relative accuracy of 5% is good enough, it should be preferred; even a drop to 1% may make a big difference in checking times or even the difference between convergence and divergence.

However, these are only rough guidelines derived from the application of a single case study and should be not automatically generalized to the analysis of other models. At least, however, they demonstrate that just using the default settings provided by PRISM is not advisable. One has to take care of various crucial settings (engine, solver, termination criterion) in order to get reasonable performance in an analysis of large-scale models or even to make the difference between the success of an analysis and its failure.

References

[1] George E. Andrews. *The Theory of Partitions*. Cambridge University Press, 1976.

[2] M. Kwiatkowska, G. Norman, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 585–591.

[3] M. Kwiatkowska, G. Norman, and D. Parker. "Stochastic Model Checking". In: *Formal Methods for Performance Evaluation: 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007*. Ed. by M. Bernardo and J. Hillston. Vol. 4486. Lecture Notes in Computer Science. Bertinoro, Italy, May 28 – June 2: Springer, 2007, pp. 220–270.

[4] David A. Parker, ed. *PRISM — Probabilistic Symbolic Model Checker*. http://www.prismmodelchecker.org. Department of Computer Science, University of Oxford, UK. 2013.

[5] Wolfgang Schreiner. *Experiments with Measuring Time in PRISM 4.0*. Technical Report. Johannes Kepler University Linz, Austria: Research Institute for Symbolic Computation (RISC), Mar. 2013. URL: http://www.risc.jku.at/publications/download/risc_4684/main.pdf.

[6] Wolfgang Schreiner. *Experiments with Measuring Time in PRISM 4.0 (Addendum)*. Technical Report. Johannes Kepler University Linz, Austria: Research Institute for Symbolic Computation (RISC), Apr. 2013. URL: http://www.risc.jku.at/publications/download/risc_4701/main.pdf.

[7] Wolfgang Schreiner. *Initial Results on Modeling in PRISM Mobile Cellular Networks with Spectrum Renting*. Technical Report. Johannes Kepler University Linz, Austria: Research Institute for Symbolic Computation (RISC), Mar. 2013. URL: http://www.risc.jku.at/publications/download/risc_4705/main.pdf.

[8] Wolfgang Schreiner, Tamas Berczes, and Janos Sztrik. *Probabilistic Model Checking on HPC Systems for the Performance Analysis of Mobile Networks*. Technical Report. Johannes Kepler University Linz, Austria: Research Institute for Symbolic Computation (RISC), Sept. 2013. URL: http://www.risc.jku.at/publications/download/risc_4819/main.pdf.

[9] Wolfgang Schreiner, Nikolaj Popov, Tamas Berczes, Janos Sztrik, and Gabor Kusper. *Applying High Performance Computing to Analyzing by Probabilistic Model Checking Mobile Cellular Networks with Spectrum Renting*. Technical Report. Johannes Kepler University Linz, Austria: Research Institute for Symbolic Computation (RISC), July 2013. URL: http://www.risc.jku.at/publications/download/risc_4705/main.pdf.

A. The PRISM Model

```
// -------------------------------------------------------------------
// Cluster24.prism
// Cluster of 24 servers to which work is distributed.
//
// Tamas Berczes <berczes.tamas@inf.unideb.hu>
// Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
// -------------------------------------------------------------------

// continuous time markov chain (ctmc) model
ctmc

// -------------------------------------------------------------------
// system parameters
// -------------------------------------------------------------------

const int N; // number of sources
const int J;  // maximum load per cluster

const double lambda; // call generation rate
const double mu;     // service rate

// -------------------------------------------------------------------
// system model
// -------------------------------------------------------------------

// the minimum load of every server
formula minServer = min(
  server1,server2,server3,server4,server5,server6,server7,server8,
  server9,server10,server11,server12,server13,server14,server15,server16,
  server17,server18,server19,server20,server21,server22,server23,server24);

// generate requests at rate sources*lambda
module Sources
  sources: [0..N] init N;

  [sservers1] sources > 0 & server1 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers2] sources > 0 & server2 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers3] sources > 0 & server3 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers4] sources > 0 & server4 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers5] sources > 0 & server5 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers6] sources > 0 & server6 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers7] sources > 0 & server7 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers8] sources > 0 & server8 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers9] sources > 0 & server9 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers10] sources > 0 & server10 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers11] sources > 0 & server11 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers12] sources > 0 & server12 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers13] sources > 0 & server13 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers14] sources > 0 & server14 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers15] sources > 0 & server15 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers16] sources > 0 & server16 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers17] sources > 0 & server17 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers18] sources > 0 & server18 = minServer ->
    sources*lambda : (sources' = sources-1);
  [sservers19] sources > 0 & server19 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers20] sources > 0 & server20 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers21] sources > 0 & server21 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers22] sources > 0 & server22 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers23] sources > 0 & server23 = minServer  ->
    sources*lambda : (sources' = sources-1);
  [sservers24] sources > 0 & server24 = minServer  ->
    sources*lambda : (sources' = sources-1);

  [ssources1] sources < N -> (sources' = sources+1);
  [ssources2] sources < N -> (sources' = sources+1);
  [ssources3] sources < N -> (sources' = sources+1);
  [ssources4] sources < N -> (sources' = sources+1);
  [ssources5] sources < N -> (sources' = sources+1);
  [ssources6] sources < N -> (sources' = sources+1);
  [ssources7] sources < N -> (sources' = sources+1);
  [ssources8] sources < N -> (sources' = sources+1);
  [ssources9] sources < N -> (sources' = sources+1);
  [ssources10] sources < N -> (sources' = sources+1);
  [ssources11] sources < N -> (sources' = sources+1);
  [ssources12] sources < N -> (sources' = sources+1);
  [ssources13] sources < N -> (sources' = sources+1);
  [ssources14] sources < N -> (sources' = sources+1);
  [ssources15] sources < N -> (sources' = sources+1);
  [ssources16] sources < N -> (sources' = sources+1);
  [ssources17] sources < N -> (sources' = sources+1);
  [ssources18] sources < N -> (sources' = sources+1);
  [ssources19] sources < N -> (sources' = sources+1);
  [ssources20] sources < N -> (sources' = sources+1);
  [ssources21] sources < N -> (sources' = sources+1);
  [ssources22] sources < N -> (sources' = sources+1);
```

```
  [ssources23] sources < N -> (sources' = sources+1);
  [ssources24] sources < N -> (sources' = sources+1);
endmodule

// serve requests at rate mu
module Server1
  server1: [0..J] init 0;
  [sservers1] server1 < J -> (server1' = server1+1);
  [ssources1] server1 > 0 -> mu : (server1' = server1-1);
endmodule

module Server2 = Server1
[server1=server2,sservers1=sservers2,ssources1=ssources2] endmodule
module Server3 = Server1
[server1=server3,sservers1=sservers3,ssources1=ssources3] endmodule
module Server4 = Server1
[server1=server4,sservers1=sservers4,ssources1=ssources4] endmodule
module Server5 = Server1
[server1=server5,sservers1=sservers5,ssources1=ssources5] endmodule
module Server6 = Server1
[server1=server6,sservers1=sservers6,ssources1=ssources6] endmodule
module Server7 = Server1
[server1=server7,sservers1=sservers7,ssources1=ssources7] endmodule
module Server8 = Server1
[server1=server8,sservers1=sservers8,ssources1=ssources8] endmodule
module Server9 = Server1
[server1=server9,sservers1=sservers9,ssources1=ssources9] endmodule
module Server10 = Server1
[server1=server10,sservers1=sservers10,ssources1=ssources10] endmodule
module Server11 = Server1
[server1=server11,sservers1=sservers11,ssources1=ssources11] endmodule
module Server12 = Server1
[server1=server12,sservers1=sservers12,ssources1=ssources12] endmodule
module Server13 = Server1
[server1=server13,sservers1=sservers13,ssources1=ssources13] endmodule
module Server14 = Server1
[server1=server14,sservers1=sservers14,ssources1=ssources14] endmodule
module Server15 = Server1
[server1=server15,sservers1=sservers15,ssources1=ssources15] endmodule
module Server16 = Server1
[server1=server16,sservers1=sservers16,ssources1=ssources16] endmodule
module Server17 = Server1
[server1=server17,sservers1=sservers17,ssources1=ssources17] endmodule
module Server18 = Server1
[server1=server18,sservers1=sservers18,ssources1=ssources18] endmodule
module Server19 = Server1
[server1=server19,sservers1=sservers19,ssources1=ssources19] endmodule
module Server20 = Server1
[server1=server20,sservers1=sservers20,ssources1=ssources20] endmodule
module Server21 = Server1
[server1=server21,sservers1=sservers21,ssources1=ssources21] endmodule
module Server22 = Server1
[server1=server22,sservers1=sservers22,ssources1=ssources22] endmodule
module Server23 = Server1
```

```
[server1=server23,sservers1=sservers23,ssources1=ssources23] endmodule
module Server24 = Server1
[server1=server24,sservers1=sservers24,ssources1=ssources24] endmodule

// ------------------------------------------------------------------
// system rewards
// ------------------------------------------------------------------

// mean number of jobs in server1
rewards "mS1"
  true : max(0, server1);
endrewards


// ------------------------------------------------------------------
// the properties to be checked
// ------------------------------------------------------------------

"mS1" : R{"mS1"}=? [ S ] ;
```