Initial Results on Modeling in PRISM Mobile Cellular Networks with Spectrum Renting*

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

April 9, 2013

Abstract

We report in this paper on our initial results on modeling in the probabilistic model checker PRISM the system described in the paper "A New Finite-Source Queueing Model for Mobile Cellular Networks Applying Spectrum Renting" by Tien v. Do et al. That paper proposes a new finite-source retrial queueing model to consider spectrum renting in mobile cellular networks; numerical results are there produced with the MOSEL-2 tool. Our results show that the model can be described and analyzed in PRISM in a very transparent way; however, due to an apparent system bug we are not yet able to check models of the same size as in MOSEL-2.

*Supported by the project HU 10/2012 of the Austrian Academic Exchange Service (ÖAD).

Contents

1. Introduction 3

2. The Model 3

3. The Properties 8

4. Conclusions 13

A. The PRISM Model and Properties 14

1. Introduction

We report on initial results to model and analyze the mobile cellular network system introduced in [2] by using the probabilistic model checker PRISM. In this model, a number of sources (cell phone subscribers) compete for access to a number of servers (channels). Sources produce requests at rate λ which a free server processes at rate μ. However, the number of available channels is not fixed: if the number of free channels gets to small, the cell phone operator may rent additional frequency blocks from another operator, partition these blocks into channels, and use the new channels for its own subscribers; these blocks may be released, if sufficiently many channels have become free again.

The model also describes the behavior of the subscribers: if they cannot get immediately access to a channel, they are offered a return call to indicate when a channel becomes available; if this offer is accepted, the subscriber request is placed into a "queue" which is served in a first-come-first-serve discipline. However, if the user becomes impatient, she may opt to leave the queue and retry the call (she is then placed into an "orbit" where calls are retried with rate ν) or to give up the call at all, upon which she is returned to the overall pool of subscribers.

In [2] the model is formalized and analyzed with the help of the MOSEL-2 tool [1]. The goal of this paper is to do the same with the probabilistic model checker PRISM [3]; for this we use the newest developer version 4.1beta available since December 2012 from the PRISM web site [4].

The rest of this paper is organized as follows: in Section 2, we describe the PRISM model and its peculiarities; in Section 3, we describe the properties to be analyzed and the results of the application of PRISM to this analysis; in Section 4, we present our (preliminary) conclusions. Appendix A lists the full source code of the model and the properties.

2. The Model

The model is a "Continuous Time Markov Chain" model introduced by the declaration

```
ctmc
```

We then introduce the model parameters

```
// renting thresholds
const int t1;      // block renting threshold
const int t2 = 6;  // block release threshold

// bounds
const int K = 50;    // population size
const int r = 8;     // number of servers/channels per block
const int m = 5;     // maximum number of blocks that can be rented
const int n = 2*r;   // minimum number of servers/channels
const int M = n+r*m; // maximum number of simultaneous calls

// rates
const double rho;                  // normalized traffic intensity
const double mu     = 1/53.22;     // service rate
const double lambda = rho*n*mu/K;  // call generation rate
const double nu     = 1;           // retrial rate
```

```
const double eta    = 1/300;      // rate of queueing users getting impatient
const double lam_r  = 1/5;        // block renting rate
const double nu_r   = 1/7;        // block rental retrial rate
const double mu_r   = 1;          // block release rate

// probabilities
const double p_b  = 0.1;       // prob. that user gives up (-> sources)
const double p_q  = 0.5;       // prob. that user presses button (-> queue)
const double p_o  = 1-p_b-p_q; // prob. that user retries later (-> orbit)
const double p_1o = 0.8;       // prob. impatient user retries (-> orbit)
const double p_1s = 1-p_1o;    // prob. impatient user gives up (-> sources)
const double p_r  = 0.8;       // block rental success probability
const double p_f  = 1-p_r;     // block rental failure probability
```

The values of these parameters are the same as those used in [2] *except* for the population size $K = 50$ (in [2], $K = 100$ is used). The reason for this deviation will be explained in Section 3.

Analogous to [2], we use an abbreviation

```
formula servAvail = n+blocks*r;
```

to denote the total number of servers (channels) that have been rented and are available for use, which depends on the value of the model variable *blocks* introduced below; in contrast to *blocks*, *servAvail* is a syntactic abbreviation whose value is not stored in the model and does therefore not contribute to the size of the state space of the system.

The model is composed of the following modules:

- *Sources* models the subscribers who are currently not performing (respectively attempting to perform) a call.
- *Servers* models the subscribers that are currently performing a call.
- *Queue* models the subscribers that are placed in the queue to wait for a return call indicating that a channel is free.
- *Orbit* models the subscribers that have opted to leave the queue and that are retrying a call on their own.

The number of subscribers in these modules is described by the corresponding model variables *sources*, *servers*, *queue*, *orbit* which satisfy the invariant $K = sources + servers + queue + orbit$. All transitions in the model describe how a subscriber moves from one module (decrementing the corresponding variable) to another module (incrementing the corresponding variable) such that the invariant is preserved. Each move is performed by a synchronized transition which is simultaneously performed by two modules:

- *sservers* moves a subscriber from *Sources* to *Servers*.
- *squeue* moves a subscriber from *Sources* to *Queue*.
- *sorbit* moves a subscriber from *Sources* to *Orbit*.
- *ssources* moves a subscriber from *Servers* to *Sources*.

- *qorbit* moves a subscriber from *Queue* to *Orbit*.
- *qservers* moves a subscriber from *Queue* to *Servers*.
- *qsources* moves a subscriber from *Queue* to *Sources*
- *osources* moves a subscriber from *Orbit* to *Sources*.

Since each transition describes a synchronized step of two modules, it is listed in two modules: in the "active" module where the move originates and in the "passive" one where the move terminates. The rate r for each transition is always indicated in the "active" module with the passive module having (implicitly) assigned rate 1. The rate of the synchronized transition is then the product $r \cdot 1 = r$.

If a module can participate in multiple transitions, guard conditions may ensure that in each state only one of the transitions is possible. If, however, in a state e.g. 2 transitions with rate r are possible, then their probabilities p_1 and $p_2 = 1 - p_1$ are used to adjust the rates of the transitions to $r \cdot p_1$ and $r \cdot p_2$.

In addition to the closely interacting modules above we have a largely independently executing module *Blocks* which describes the management of the frequency blocks with a variable *blocks* indicating the number of rented blocks. The management of the blocks is performed by the following transitions:

- *success* successfully rents a new block.
- *failure* fails to rent a block.
- *retrial* returns after a failed attempt to rent a block to a state where the module retries to rent a block.
- *interrupt* returns after a failed attempt to rent a block to a state where the module does not retry to rent a block.
- *release* returns a rented block.

In more detail, the modules are described below.

Sources This module has one system variable *sources* whose value is in interval $[0, K]$ which is managed by three "active" transitions *sservers*, *sorbit*, and *squeue*, and three "passive" ones, *ssources*, *osources*, *qsources*.

```
module Sources
  sources: [0..K] init K;
  [sservers] sources > 0 & servers < servAvail ->
    sources*lambda : (sources' = sources-1);
  [sorbit]   sources > 0 & servers = servAvail ->
    sources*lambda*p_o : (sources' = sources-1);
  [squeue]   sources > 0 & servers = servAvail ->
    sources*lambda*p_q : (sources' = sources-1);
  [ssources] sources < K -> (sources' = sources+1);
```

```
    [osources] sources < K -> (sources' = sources+1);
    [qsources] sources < K -> (sources' = sources+1);
  endmodule
```

Among the active transitions, if $servers < servAvail$, only *sservers* is enabled; if $servers = servAvail$, transition *sorbit* is taken with probability p_o, transition *squeue* is taken with probability p_r; with probability $p_b = 1 - p_o - p_r$, the module stays in its state (which need not be indicated explicitly).

In the passive transitions, the condition $sources < K$ might not seem necessary, because in every reachable system state the transition will only be attempted, when this condition is ensured; however, if we omit the condition, PRISM will complain, because it checks the validity of the module with respect to every possible state (also those that are not reachable).

This module differs somewhat from the original formulation in MOSEL-2 where subscribers from *Sources* were moved first into a state *Requests* in which the decision was made, whether it is possible to enter *Servers* or whether the user may be offered to enter the *Queue* or the *Orbit*. However, this intermediate state is only *virtual*, in the sense that the transition from *Sources* to *Requests* takes zero time, i.e., it does not correspond to a transition in the real system. Since there are no zero-time transitions in PRISM, we omit this intermediate state.

Servers This module has one system variable *servers* which is managed by one active transition *ssources* and two passive transitions *sservers* and *qservers*.

```
module Servers
  servers: [0..M] init 0;
  [sservers] servers < M -> (servers' = servers+1);
  [qservers] servers < M -> (servers' = servers+1);
  [ssources] servers > 0 -> servers*mu : (servers' = servers-1);
endmodule
```

Again the passive transitions are guarded by the redundant condition $servers < M$, to avoid PRISM complaining.

Queue this model has one system variable *queue* which is managed by the active transitions *qservers*, *qorbit*, and *qsources*, and the passive transition *squeue*:

```
module Queue
  queue: [0..K-n] init 0;
  [squeue]   queue < K-n -> (queue' = queue+1);
  [qservers] queue > 0 & servers < servAvail ->
    9999 : (queue' = queue-1); // "immediately"
  [qorbit]   queue > 0 & servers = servAvail ->
    queue*eta*p_1o : (queue' = queue-1);
  [qsources] queue > 0 & servers = servAvail ->
    queue*eta*p_1s : (queue' = queue-1);
endmodule
```

The passive transition is guarded by the redundant condition $queue < K - n$ to avoid complaints of PRISM. Among the active transitions, in a state with $servers < servAvail$, only *qservers*

is enabled which moves a message from *Queue* to *Servers*. In the MOSEL-2 model, this transition is performed with time 0; in the PRISM model, we reflect this by an "infinite" rate 9999. However, we consider this "zero-time" transition a deficiency of the MOSEL-2 model and have contacted one of the authors to reconsider that aspect.

In a state with $servers = servAvail$, transitions *qorbit* and *qsources* are taken with probability p_{1o} and $p_{1s} = 1 - p_{1o}$, respectively. In [2], however this condition is missing, which may or may not indicate a deficiency of the model; we have asked one of the authors for clarification.

Orbit This module encapsulates a system variable *orbit* with active transition *osources* and passive transitions *sorbit* and *qorbit* (which are guarded by the redundant conditions $orbit < K - n$ to avoid PRISM complaints).

```
formula orbit = K-(sources+servers+queue); // make variable virtual
module Orbit
  // orbit: [0..K-n] init 0;
  [sorbit]   orbit < K-n -> true; // (orbit' = orbit+1);
  [qorbit]   orbit < K-n -> true; // (orbit' = orbit+1);
  [osources] orbit > 0   -> orbit*nu : true; // (orbit' = orbit-1);
endmodule
```

However, rather than modeling *orbit* by an explicit state variable, we remember our system invariant $K = sources + servers + queue + orbit$, from which we can deduce that $orbit = K - (sources + servers + queue)$, i.e., the value of *orbit* can be deduced from the other system variables. Since the size of the state space of a system is the crucial factor for the time needed for checking system properties, we therefore decided to make this variable *virtual*, i.e., replace it by the term $K - (sources + servers + queue)$. Model checking time was thus decreased by one order of magnitude (approximately a factor of 10).

Blocks This module encapsulates two state variables *blocks* and *trial* where *blocks* models the number of rented frequency blocks and *trial* becomes 1 by a failed attempt to rent a block.

```
module Blocks
  blocks: [0..m] init 0;
  trial: [0..1] init 0;
  [success] trial = 0 & servAvail-servers <= t1 & blocks < m ->
    lam_r*p_r: (blocks' = blocks+1);
  [failure] trial = 0 & servAvail-servers <= t1 & blocks < m ->
    lam_r*p_f: (trial' = 1);
  [retrial] trial = 1 & servAvail-servers <= t1 ->
    nu_r : (trial' = 0);
  [interrupt] trial = 1 & servAvail-servers > t1  ->
    9999 : (trial' = 0); // "immediately"
  [release] servAvail-servers >= t2+r & blocks > 0 ->
    mu_r : (blocks' = blocks-1);
endmodule
```

If it is decided that an attempt is to be made to rent a block ($servAvail - servers \leq t_1$), then this attempt succeeds with probability p_r and fails with probability $p_f = 1 - p_r$. In the later

case, we enter state $trial = 1$, from which another attempt will be made only after returning to state $trial = 0$. The transition from state $trial = 1$ to $trial = 0$ is made with rate ν_r unless, it is detected that a retrial is not necessary ($servAvail_servers > t_1$); in this case, the transition is taken immediately (with "infinite" rate 9999). If it is detected, that sufficiently many blocks are available ($servAvail - servers \geq t_2 + r$), a block is released.

Above module differs from the one presented in[2] which moves from a state where it has been decided that a block should be allocated by a "zero-time" transition to a state *Blocks*, if the attempt is successful, respectively to state *RentOrbit* where the retrial shall be attempted. We avoid this transition in our model above.

However, we did not see how to model the "Interruption" that may take in a retrial state except by adding the "zero-time" transition *interrupt* indicated above.

The Order of Modules While logically the order of modules in a PRISM model is of no significance, it may have a major impact on model checking time: every state is represented by a bit vector where the variables are represented in the order in which they are listed in the model (i.e., in the order of the modules). States and state sets are respresented in PRISM by binary decision diagrams (BDDs) where the time required for specific operations is sensitive to the order of bits by which a state is represented. As a general strategy, "important" variables should occur first.

From an original model, where module *Server* was listed last to one where it was listed first, we observed a speedup by a factor of 2–4; after some experiments, we deemed the order of modules listed in Section A as the most efficient one.

3. The Properties

All the properties listed in [2] can be formulated with the help of a number of state rewards, i.e., values assigned to every system state (the term $\max(0, orbit)$ used below only required because of the "virtual" definition of *orbit* which for unreachable states may yield negative values, such that PRISM complains):

```
// mean number of active requests
rewards "mM"
  true : max(0, orbit)+queue+servers;
endrewards

// mean number of calls in orbit
rewards "mO"
  true : max(0, orbit);
endrewards

// mean number of calls in queue
rewards "mQ"
  true: queue;
endrewards

// mean number of active calls
```

```
rewards "mC"
  true: servers;
endrewards

// mean number of active blocks
rewards "mB"
  true: blocks;
endrewards
```

The corresponding queries can be then performed as follows, very much in the style also shown in [2]:

```
/ mean number of active requests
"mM" : R{"mM"}=? [ S ] ;

// mean number of active sources
"mK" : K-"mM" ;

// mean throughput (served and unserved)
"m1" : "mK"*lambda ;

// mean number of active calls
"mC" : R{"mC"}=? [ S ] ;

// mean goodput
"m1good" : "mC"*mu ;

// probability that arriving customer gets served
"Pgood" : "m1good"/"m1" ;

// mean response time (served and unserved)
"mT" : "mM"/"m1" ;

// mean number of rented blocks
"mB" : R{"mB"}=? [ S ] ;

// mean number of available servers
"mS" : n+"mB"*r ;

// mean number of idle servers
"mAS" : "mS"-"mC" ;

// utilization of available servers
"Sutil" : "mC"/"mS" ;

// blocking probability
"Pblock" : S=? [ servers = servAvail ] ;

const int B;

// probability that B blocks are partially utilized
"Pb" : S=? [ n+r*(B-1) < servers & servers <= n+r*B ] ;

// mean queue length
"mQ" : R{"mQ"}=? [ S ] ;

// mean time spent in queue
"mTQ" : "mQ" / "m1" ;

// mean orbit length
```

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

// mean time spent in orbit
"mTO" : "mO" / "m1" ;
```

These queries depend only on calculation of steady state rewards such as

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

and steady state probabilites such as

```
"Pblock" : S=? [ servers = servAvail ] ;
```

In the computation of times (e.g., query “mTO”), Little’s Law is applied; as discussed in [5], we may investigate in the future also explicit time measurement for this purpose.

As for the actual application of PRISM to check these probabilities, we found it most efficient to use the “Sparse” engine and the “Gauss-Seidel” solver; for larger models, it was necessary to increase the maximum memory size allocated to the CUDD package for manipulating binary decision diagrams from its default of 400 MB to more than 1 GB; otherwise the model creation would fail with an error message (a segmentation violation of the Java process, corresponding hints can be found in the FAQ section of the PRISM web site [4]).

With these settings and the optimization of “virtualizing” the variable *orbit* as described in Section 2, it is possible to check models for $K = 50$ in a few (2–4) seconds; without the virtualization, it took more than half a minute). Since increasing K by a value of 10 approximately doubles the model checking time, we could expect checking for $K = 100$ in a couple of minutes (which seems significantly slower than MOSEL which needs for a check about half a minute).

However, when increasing the parameter to $K \geq 60$, we encountered the problem that the setup of the linear equation system for the “Sparse” engine which looks for $K = 50$ like

```
Building sparse matrix... [n=80678, nnz=381238, compact] [1.5 MB]
Creating vector for diagonals... [dist=40401, compact] [473.2 KB]
Allocating iteration vector... [630.3 KB]
TOTAL: [2.6 MB]
```

for $K = 60$ stops at the point

```
Building sparse matrix... [n=166513, nnz=803018, compact] [3.2 MB]
Creating vector for diagonals...
```

with the process continuing to consume CPU time but not producing any output (or error); we suspect this to be a bug in PRISM and have informed the developers of PRISM correspondingly.

Since this bug prevents the use of the “Sparse” and also of the “Hybrid” engine (and the other engines “MTBDD” and “Explicit” were much too slow), we were only able ro run preliminary tests for $K = 50$, rather than for $K = 100$ as in [2]. The results are shown in Figures 1 and 2, which depict for $K = 50$ the results shown for $K = 100$ in Figures 3 and 4 of [2].

The diagram for “Mean Number of Channels” in Figure 1 and all the diagrams in Figure 2 look qualitatively similar (with different quantitative values) as the corresponding ones in [2]. However, while the other diagrams in Figure 1 have similar trends, they do not form smooth monotonously growing curves like those depicted in [2], but depict some “down” bumps when ρ is a multiple of 1.6. We are not sure whether this corresponds to the original model or not; we have asked one of the authors of [2] for corresponding diagrams for $K = 50$.

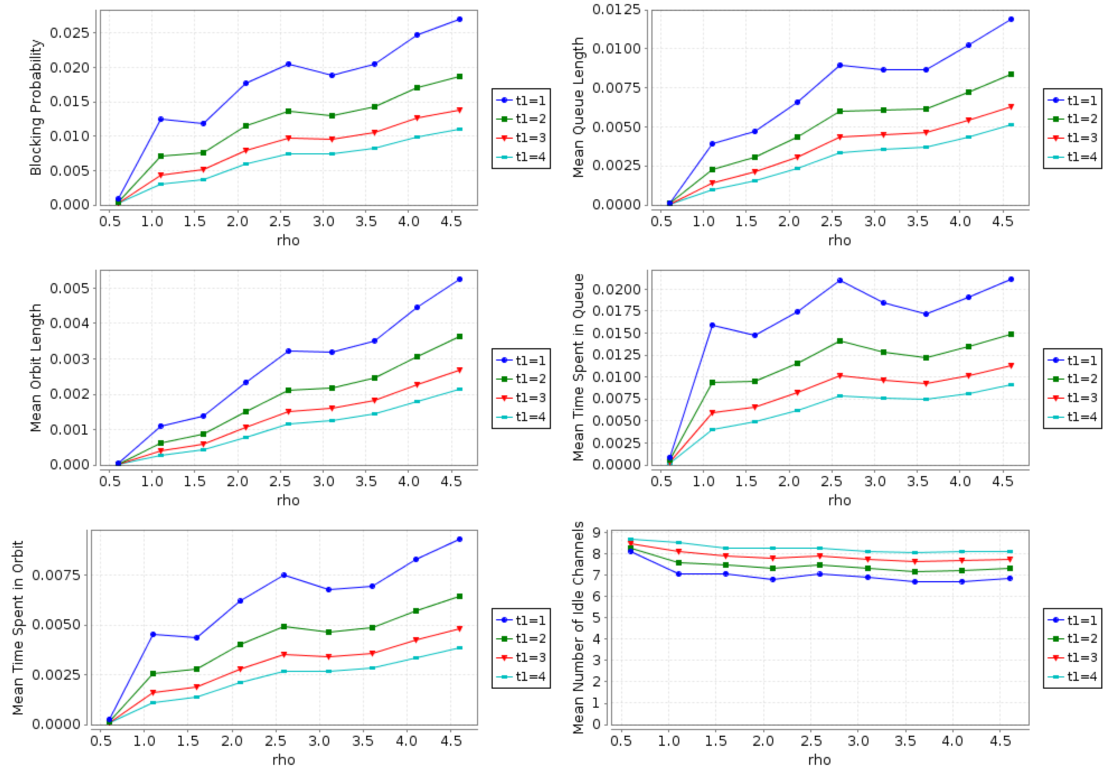

Figure 1: Performance Measures for $t_2 = 6$

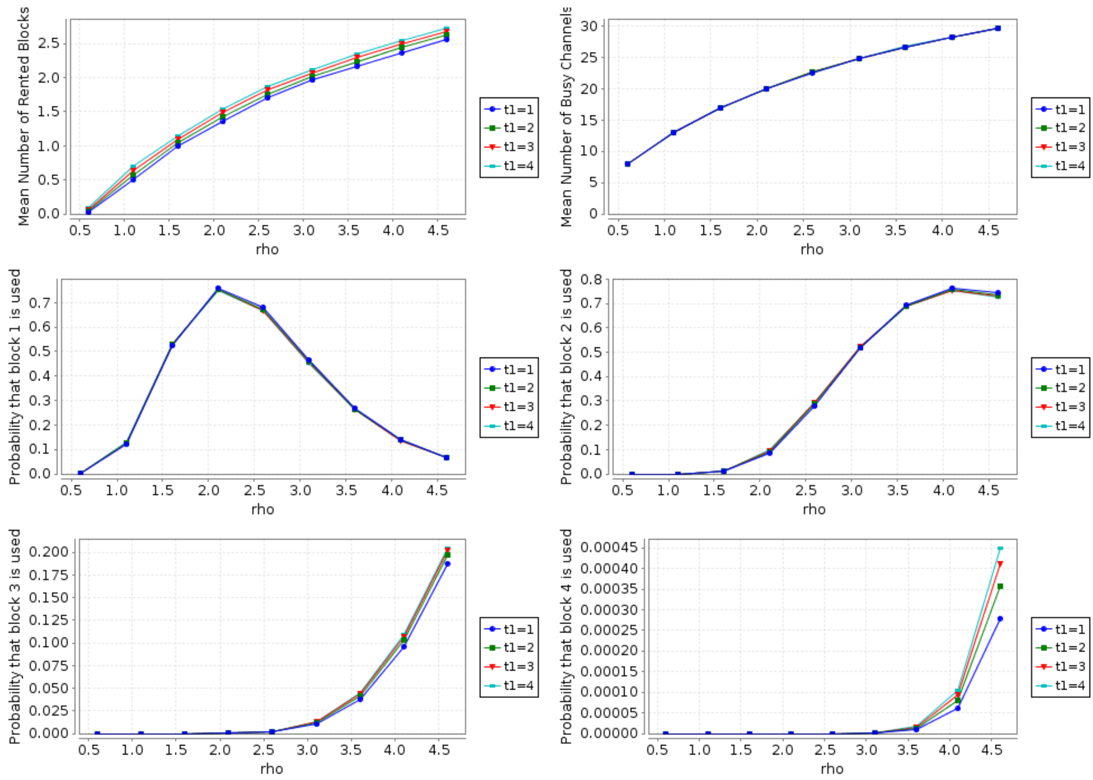

Figure 2: Performance Measures for $t_2 = 6$

4. Conclusions

As shown in this paper, it seems feasible to create and model in PRISM 4.1 systems analogously to those presented in [2] using MOSEL-2. The PRISM models seem to us more transparent than that in MOSEL because they correspond more directly to the intuition; essentially each block depicted in Figure 1 of [2] becomes one PRISM module and the interaction between the blocks become shared transitions of the corresponding modules forwarding values from one module to the other.

We also identified some aspects of the MOSEL model (a zero time transition between the "Queue" component and the "Server" component and an apparently missing guard condition in the "Queue" model) which seem questionable to us and about which we have asked one of the authors of [2] for clarification. There is also one aspect of our model (a zero-time transition in module "Blocks") which we do not like particularly well and would like to revise.

As for analyzing the model, we saw that all the queries performed in MOSEL can be performed in an analogous way in PRISM (with the possibility of explicit time measurement as described in [5] still to be investigated). For the model checking time, however, we had to perform quite some optimizations (considering the order of system modules/variables, virtualizing a system variable) to get in a similar order of magnitude than the checks performed by MOSEL.

Unfortunately, we ultimately failed to analyze models of the same size as in [2] due to an apparent bug in the PRISM model checker, about which we have contacted the PRISM developers. Until the bug is fixed or the authors of [2] produce numerical results for smaller models, we cannot be sure that our models yield the same results. While this seems to be the case at least for some of the queries, for other ones the outcome is still questionable.

References

[1] K. Begain, G. Bolch, and Herold H. *Practical Performance Modeling Application of the MOSEL Language*. Kluwer Academic Publisher, 2012.

[2] Tien v. Do, Patrick Wüchner, Tamas Berczes, Janos Sztrik, and Hermann de Meer. "A New Finite-Source Queueing Model for Mobile Cellular Networks Applying Spectrum Renting". In: *Asia-Pacific Journal of Operational Research* (2013). To appear.

[3] 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.

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

[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`.

A. The PRISM Model and Properties

```
// ----------------------------------------------------------------------
// Spectrum.prism
// A model for mobile cellular networks applying spectrum renting.
//
// The model is described in
//
//   Tien v. Do, Patrick Wüchner, Tamas Berczes, Janos Sztrik,
//   Hermann de Meer: A New Finite-Source Queueing Model for
//   Mobile Cellular Networks Applying Spectrum Renting,
//   September 2012.
//
// Use for fastest checking the "Sparse" engine and the "Gauss-Seidel" solver;
// for larger K, it may be necessary to increase the CUDD maximum memory size
// to more than 1 GB, otherwise model construction fails.
//
// Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
// Copyright (C) 2013, Research Institute for Symbolic Computation
// Johannes Kepler University, Linz, Austria, http://www.risc.jku.at
// ----------------------------------------------------------------------

// continuous time markov chain (ctmc) model
ctmc

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

// renting thresholds
const int t1;      // block renting threshold
const int t2 = 6;  // block release threshold

// bounds
const int K = 50;    // population size
const int r = 8;     // number of servers/channels per block
const int m = 5;     // maximum number of blocks that can be rented
const int n = 2*r;   // minimum number of servers/channels
const int M = n+r*m; // maximum number of simultaneous calls

// rates
const double rho;                  // normalized traffic intensity
const double mu     = 1/53.22;     // service rate
const double lambda = rho*n*mu/K;  // call generation rate
const double nu     = 1;           // retrial rate
const double eta    = 1/300;       // rate of queueing users getting impatient
const double lam_r  = 1/5;         // block renting rate
const double nu_r   = 1/7;         // block rental retrial rate
const double mu_r   = 1;           // block release rate

// probabilities
const double p_b  = 0.1;       // prob. that user gives up (-> sources)
const double p_q  = 0.5;       // prob. that user presses button (-> queue)
const double p_o  = 1-p_b-p_q; // prob. that user retries later (-> orbit)
```

```
const double p_1o = 0.8;       // prob. that impatient user retries later (-> orbit)
const double p_1s = 1-p_1o;    // prob. that impatient user gives up (-> sources)
const double p_r  = 0.8;       // block rental success probability
const double p_f  = 1-p_r;     // block rental failure probability

// ------------------------------------------------------------------
// system model
// note that the order of the modules influences model checking time
// heuristically, this seems to be the best one
// ------------------------------------------------------------------

// number of currently available servers/channels
formula servAvail = n+blocks*r;

// blocks are rented at rate lam_r and released at rate mu_r
// renting is successful with probability p_r and fails with probability p_f
// retrying a failed attempt is performed at rate nu_r
module Blocks
  blocks: [0..m] init 0;
  trial: [0..1] init 0;
  [success] trial = 0 & servAvail-servers <= t1 & blocks < m ->
    lam_r*p_r: (blocks' = blocks+1);
  [failure] trial = 0 & servAvail-servers <= t1 & blocks < m ->
    lam_r*p_f: (trial' = 1);
  [retrial] trial = 1 & servAvail-servers <= t1 ->
    nu_r : (trial' = 0);
  [interrupt] trial = 1 & servAvail-servers > t1  ->
    9999 : (trial' = 0); // "immediately"
  [release] servAvail-servers >= t2+r & blocks > 0 ->
    mu_r : (blocks' = blocks-1);
endmodule

// available servers accept requests
module Servers
  servers: [0..M] init 0;
  [sservers] servers < M -> (servers' = servers+1);
  [qservers] servers < M -> (servers' = servers+1);
  [ssources] servers > 0 -> servers*mu : (servers' = servers-1);
endmodule

// generate requests at rate sources*lambda
module Sources
  sources: [0..K] init K;
  [sservers] sources > 0 & servers < servAvail ->
    sources*lambda : (sources' = sources-1);
  [sorbit]   sources > 0 & servers = servAvail ->
    sources*lambda*p_o : (sources' = sources-1);
  [squeue]   sources > 0 & servers = servAvail ->
    sources*lambda*p_q : (sources' = sources-1);
  [ssources] sources < K -> (sources' = sources+1);
  [osources] sources < K -> (sources' = sources+1);
  [qsources] sources < K -> (sources' = sources+1);
endmodule
```

```
// if no server is available, requests are redirected
// with probability p_o to the orbit
formula orbit = K-(sources+servers+queue); // make variable virtual
module Orbit
  // orbit: [0..K-n] init 0;
  [sorbit]   orbit < K-n -> true; // (orbit' = orbit+1);
  [qorbit]   orbit < K-n -> true; // (orbit' = orbit+1);
  [osources] orbit > 0   -> orbit*nu : true; // (orbit' = orbit-1);
endmodule

// if no server is available, requests are redirected
// with probability p_q to the queue
module Queue
  queue: [0..K-n] init 0;
  [squeue]   queue < K-n -> (queue' = queue+1);
  [qservers] queue > 0 & servers < servAvail ->
    9999 : (queue' = queue-1); // "immediately"
  [qorbit]   queue > 0 & servers = servAvail ->
    queue*eta*p_1o : (queue' = queue-1);
  [qsources] queue > 0 & servers = servAvail ->
    queue*eta*p_1s : (queue' = queue-1);
endmodule

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

// mean number of active requests
rewards "mM"
  true : max(0, orbit)+queue+servers;
endrewards

// mean number of calls in orbit
rewards "mO"
  true : max(0, orbit);
endrewards

// mean number of calls in queue
rewards "mQ"
  true: queue;
endrewards

// mean number of active calls
rewards "mC"
  true: servers;
endrewards

// mean number of active blocks
rewards "mB"
  true: blocks;
endrewards


// ------------------------------------------------------------------
```

```
// Spectrum.props
// -----------------------------------------------------------------

// mean number of active requests
"mM" : R{"mM"}=? [ S ] ;

// mean number of active sources
"mK" : K-"mM" ;

// mean throughput (served and unserved)
"m1" : "mK"*lambda ;

// mean number of active calls
"mC" : R{"mC"}=? [ S ] ;

// mean goodput
"m1good" : "mC"*mu ;

// probability that arriving customer gets served
"Pgood" : "m1good"/"m1" ;

// mean response time (served and unserved)
"mT" : "mM"/"m1" ;

// mean number of rented blocks
"mB" : R{"mB"}=? [ S ] ;

// mean number of available servers
"mS" : n+"mB"*r ;

// mean number of idle servers
"mAS" : "mS"-"mC" ;

// utilization of available servers
"Sutil" : "mC"/"mS" ;

// blocking probability
"Pblock" : S=? [ servers = servAvail ] ;

const int B;

// probability that B blocks are partially utilized
"Pb" : S=? [ n+r*(B-1) < servers & servers <= n+r*B ] ;

// mean queue length
"mQ" : R{"mQ"}=? [ S ] ;

// mean time spent in queue
"mTQ" : "mQ" / "m1" ;

// mean orbit length
"mO" : R{"mO"}=? [ S ] ;

// mean time spent in orbit
```

```
"mTO" : "mO" / "m1" ;
```