Modeling RF Communication in Sensor Networks by Probabilistic Model Checking*

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

Tamás Bérczes*
berczes.tamas@inf.unideb.hu

János Sztrik*
berczes.tamas@inf.unideb.hu

Ádám Tóth*
adamtoth102@gmail.com

* Faculty of Informatics
University of Debrecen, Hungary

October 6, 2015

Abstract

We report in this paper our results of modeling and analyzing with the probabilistic model checker PRISM a system of radio frequency (RF) transmission in sensor networks which has previously been studied in literature by using finite-source retrial queueing systems. We are able to validate with a small and quite transparent PRISM model the previously reported results (and also exhibit a minor error). Furthermore, we extend the model by also considering infinite sources and show that a previously suggested optimization has in this model beneficial effects only in a comparatively small parameter range.

*Supported by the project 90öu6 "Leistungsmodellierung von Drahtlosen Sensor-Netzwerken" of the Stiftung Aktion Österreich-Ungarn.

Contents

1 Introduction

The goal of this paper is to validate and extend the analysis of a model of radio frequency (RF) communication in wireless sensor networks. The original version of the model has been described in [3] and analyzed by using finite-source retrial queueing systems (a shorter earlier variant of the paper has appeared in [2]). The model consists of the following main components:

- A radio frequency RF unit transmits data recorded by two classes of sensors, "normal" sensors and "emergency" sensors.
- Transmission requests from emergency sensors enter a common queue: as long as this queue is not empty, the RF unit handles emergency requests from the queue.
- If there are no more emergency requests in the queue, the RF unit also accepts transmission requests from normal sensors. If a normal sensor detects that its request is not accepted by the RF unit, the sensor becomes idle and retries its request after some time again.
- Both emergency and normal sensors can only request a new transmission after the RF unit has accepted and completed the previous transmission of the respective sensor.
- In order to save power, the RF may switch from its energy-consuming "on" state to an energy-efficient "off" state. The unit turns "off", if for a certain period of time there have not been any transmission requests from either class of sensors; after some time the unit turns "on" again to check for new transmission requests.
- Optionally, the system operates in a mode where an emergency request may prematurely "wake up" the RF unit from its "off" state.

In [3] the system was numerically analyzed with the performance evaluation tool MOSEL [1]; it was shown that the optional "wake up" mode improves the characteristics of the system by reducing the response time without substantially increasing the idle time of the RF unit.

The goal of this paper is to validate and extend these results with the help of the probabilistic model checker PRISM [5, 4]:

- We construct a new PRISM model for the system which is able to reproduce most of the numerical results presented in [3]; however, also some (minor) errors in the original presentation are revealed.
- We extend the original analysis by also considering a variant of the model where a normal sensor can request a new transmission without waiting for the completion of its previous transmission request.

The remainder of the paper is organized as follows: In Section 2 we introduce and analyze the PRISM model which corresponds to the previously published model. In Section 3, we present and analyze the extended version of the model. In Section 4 we present our conclusions and discuss further directions. Appendices A and B list the complete PRISM source code of the models and the queries used for its analysis.

2 Finite Source Model

The PRISM model uses the following parameters:

```
const int N = 50;  // number of emergency sensors
const int K = 50;  // number of standard sensors

const double lambda;               // overall generation rate [0.1,4.6]
const double lambda1 = lambda*0.1; // emergency generation rate
const double lambda2 = lambda*0.9; // standard generation rate

const double nu = 2;     // retrial rate
const double mu = 20;    // service rate
const double gamma = 10; // initialization rate

const double alpha1; // =1/alpha: mean time of listening period [0.1,2] or 1.5
const double beta1;  // =1/beta:  mean time of sleeping period [0.5,2.5]

const int wakeup; // 1: emergency request wakes up RF unit
```

The parameters N and K denote the number of emergency sensors and standard sensors, respectively. The parameter λ denotes the total rate at which transmission requests are generated (the execution times of all state transitions in the model are assumed to be exponentially distributed). $\lambda_1 = 0.1 \cdot \lambda$ represents the rate of transmission requests of emergency sensors while $\lambda_2 = 0.9 \cdot \lambda$ represents the rate of transmission requests of normal sensors. The inverse of rate ν denotes the mean time after which a normal sensor retries to transmit a request, if its previous attempt has not been honored. The inverse of rate μ denotes the mean time for a data transmission of the RF unit. The inverse of rate γ denotes the mean time for waking up the RF unit in the optional "wake up" mode (which is enabled if the parameter *wakeup* is set to 1). The parameter α_1 indicates the mean time the RF unit waits for new requests before going to its "off" state; the parameter β_1 indicates the mean time the RF unit is in "off" state before becoming "on" again to check for new requests.

The generation of requests from emergency sensors are handled by the following components of the model:

```
module Emergency
  k1: [0..N] init N;
  [equeue] k1 > 0 -> k1*lambda1 : (k1' = k1-1);
  [edone]  k1 < N -> (k1' = k1+1);
endmodule

module Queue
  q: [0..N] init 0;
  [equeue]  q < N -> (q' = q+1);
  [qserver] q > 0 -> (q' = q-1);
endmodule
```

Module *Emergency* encapsulates the number k_1, $0 \leq k_1 \leq N$, of emergency sensors that are currently "active" (recording data); whenever a new measurement has been taken it is forwarded by the synchronized action *equeue* to the module *Queue* which encapsulates the number q,

$0 \leq q \leq N$, of pending transmission requests from emergency sensors. Since there are k_1 active sensors, this action happens with rate $k_1 \cdot \lambda_1$.

The generation of requests from normal sensors are handled by the following components of the model:

```
module Normal
  k2: [0..K] init K;
  [nserver] k2 > 0 -> k2*lambda2 : (k2' = k2-1);
  [norbit]  k2 > 0 -> k2*lambda2 : (k2' = k2-1);
  [ndone]   k2 < K -> (k2' = k2+1);
endmodule

module Orbit
  o: [0..K] init 0;
  [norbit]  o < K & !(ison = true & job = 0 & q = 0) -> (o' = o+1);
  [oserver] o > 0 -> o*nu : (o' = o-1);
endmodule
```

Module *Normal* encapsulates the number k_2, $0 \leq k_2 \leq K$, of normal sensors that are currently "active" (recording data). Whenever a new measurement has been taken, it may be forwarded to the RF unit by the synchronized action *nserver*, if the unit accepts the request (see below). Otherwise the request is forwarded by the synchronized action *norbit* to the module *Orbit* which encapsulates the number o, $0 \leq q \leq K$, of normal sensors that are waiting for the RF unit to accept their request. The precondition

```
!(ison = true & job = 0 & q = 0)
```

of the synchronized action *norbit* is the negation of the precondition of the synchronized action *nserver* by which a request is forwarded to the RF unit (see below). Therefore at any time only one of the actions is enabled. Since there are k_2 active sensors, each of the two actions happens with rate $k_2 \cdot \lambda_2$.

The operation of the RF unit is described by the following component:

```
module Server
  ison: bool init true;
  job: [0..2] init 0; // 1: normal job, 2: emergency job
  [switchoff] ison = true & job = 0 & q = 0 -> 1/alpha1 : (ison' = false);
  [switchon]  ison = false -> 1/beta1 : (ison' = true);
  [wakeup]    ison = false & q > 0 & wakeup = 1 -> gamma : (ison' = true);
  [nserver]   ison = true & job = 0 & q = 0 -> (job' = 1);
  [oserver]   ison = true & job = 0 & q = 0 -> (job' = 1);
  [ndone]     job = 1 -> mu : (job' = 0);
  [qserver]   ison = true & job = 0 -> infinity : (job' = 2);
  [edone]     job = 2 -> mu : (job' = 0);
endmodule
```

The state of module *Server* is described by two variables: The Boolean flag *ison* indicates whether the RF unit is in state "on" or "off". The variable *job* indicates which kind of data transmission (if any) is currently processed: value 0 indicates no job, value 1 indicates a transmission from a normal sensor, value 2 indicates a transmission from an emergency sensor. In more detail, we have the following actions:

- *switchoff*: if the RF unit is "on" but there is no job processed and none in the emergency queue, the unit switches off after mean time α_1.
- *switchon*: if the RF unit is "off", the unit switches on again after mean time β_1.
- *wakeup*: if the RF unit is "off", there is a job in the emergency queue, and the optional "wakeup" mode is selected, the RF unit is switched on after the mean initialization time γ.
- *nserver*: this action (which is synchronized with the corresponding action in the module *Normal*) accepts a transmission from a normal sensor, if the RF unit is "on", idle, and there is no emergency transmission in the queue.
- *oserver*: this action (which is synchronized with the corresponding action in the module *Orbit*) accepts a transmission from a normal sensor whose previous transmission attempt was unsuccessful;
- *ndone*: this action (which is synchronized with the corresponding action in the module *Normal*) completes the transmission of data from a normal sensor in mean time $1/\mu$, thus freeing the RF unit, and enabling another transmission of the sensor.
- *qserver*: this action (which is synchronized with the corresponding action in the module *Queue*) accepts a transmission from the emergency queue, provided the RF unit is on and idle; we assume that this happens in "zero" time, i.e., with rate ∞.
- *edone*: this action (which is synchronized with the corresponding action in the module *Emergency*) completes the transmission of data from an emergency sensor in mean time $1/\mu$, thus freeing the RF unit, and enabling another transmission of the sensor.

In order to formulate the necessary PRISM queries we equip the model with a number of "reward structures" that associate to every state certain numerical values.

```
rewards "qlength"
  true : q;
endrewards

rewards "osize"
  true : o;
endrewards

rewards "eactive"
  true : k1;
endrewards

rewards "nactive"
  true : k2;
endrewards

rewards "sleeping"
  !ison : 1;
endrewards
```

```
rewards "idle"
  ison & job = 0: 1;
endrewards

rewards "busy"
  ison & job != 0: 1;
endrewards
```

In detail, *qlength* captures the number of transmission requests in the emergency queue, *osize* denotes the number of normal sensors that are idling by waiting for the transmission of their data, *eactive* and *nactive* denote the number of active emergency sensors and normal sensors, respectively, *sleeping* indicates the probability of the RF unit being in the "off" state, *idle* indicates the probability of the unit being in the "on" state without actually transmitting data, *busy* indicates the probability of the unit transmitting data.

While this model (listed in Appendix A.1) produces (as shown below) the required results, it suffers from a certain "inelegance" by a transition with an infinite rate required by forwarding a transmission request from the module *Queue* to the module *Server* before actually processing the request. We therefore provide an alternative version of the system which copes without this intermediate state transition by the following modifications:

```
module Queue
  q: [0..N] init 0;
  [equeue] q < N -> (q' = q+1);
  [edone]  q > 0 -> (q' = q-1);
endmodule

module Server
  ison: bool init true;
  job: [0..1] init 0; // 1: normal job
  ...
  [ndone]     job = 1 -> mu : (job' = 0);
  [edone]     ison = true & job = 0 -> mu : true;
endmodule
```

In this version of the model (listed in Appendix A.2), there is no action *qserver*, instead the action *edone* is synchronized between the three modules *Emergency*, *Queue*, and *Server*: the action is enabled, if there is a transmission request in *Queue* while the RF unit is "on" and idle; it removes with rate μ the request from *Queue* and enables the emergency sensor for another transmission; the state of the RF unit itself is not changed by the action, i.e., the variable *job* does not take value 2 any more.

This new model differs from the original one in that the queue now also contains the emergency request that is currently processed by the RF unit. However by modifying the reward structure as follows

```
rewards "qlength"
    ison & job = 0 & q > 0  : q-1;
  !(ison & job = 0 & q > 0) : q;
endrewards
```

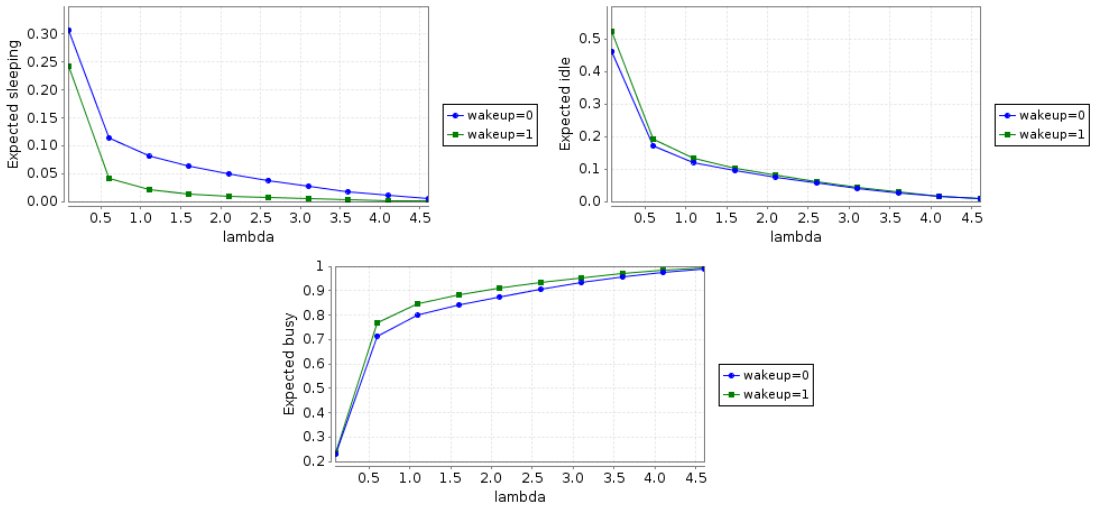

Figure 1: Finite Source: Probabilities ($\lambda = 0.5, \alpha_1 = 1/\alpha = 1.5, \beta_1 = 1/\beta = 1.0$)

we compensate this difference: if the RF unit is currently processing an emergency request (which is the case if the unit is "on", there is such a request in the queue, and there is no request from a normal sensor being processed), we subtract 1 from q to determine the number of emergency requests that are actually waiting in the queue for being processed.

For either of these models, we can now reproduce by the PRISM queries

```
"qlength":  R{"qlength"}=? [ S ]
"osize":    R{"osize"}=? [ S ]
"eactive":  R{"eactive"}=? [ S ]
"nactive":  R{"nactive"}=? [ S ]
"qtime":    "qlength"/(lambda1*"eactive");
"otime":    "osize"/(lambda2*"nactive");
"sleeping": R{"sleeping"}=? [ S ]
"idle":     R{"idle"}=? [ S ]
"busy":     R{"busy"}=? [ S ]
```

the results reported in [3]: most of these queries just ask for the expected values of the corresponding reward structures in the long term ("steady state rewards"); the queries *qtime* and *otime* apply "Little's Law" to determine from the expected number of transmission requests in the queue respectively in the orbit, from the request generation rates, and from the number of active servers, the expected time that requests spend in the queue respectively orbit.

Using the PRISM model checker (applying the "sparse" engine and the "Jacobi" solver, each data point can be computed in 1–2 seconds), the following results were produced[1]:

- Figure 1 reproduces the probabilities given in Figures 9–11 of [3];
- the top 6 diagrams of Figure 2 reproduces the quantitative measures given in Figures 3–8 of [3];

[1] If the diagrams do not list other values, we use as in [3] the parameter values $\lambda = 0.5, \alpha_1 = 1/\alpha = 1.5, \beta_1 = 1/\beta = 1.0$; actually the value for β_1 is missing in [3] but was experimentally reconstructed.

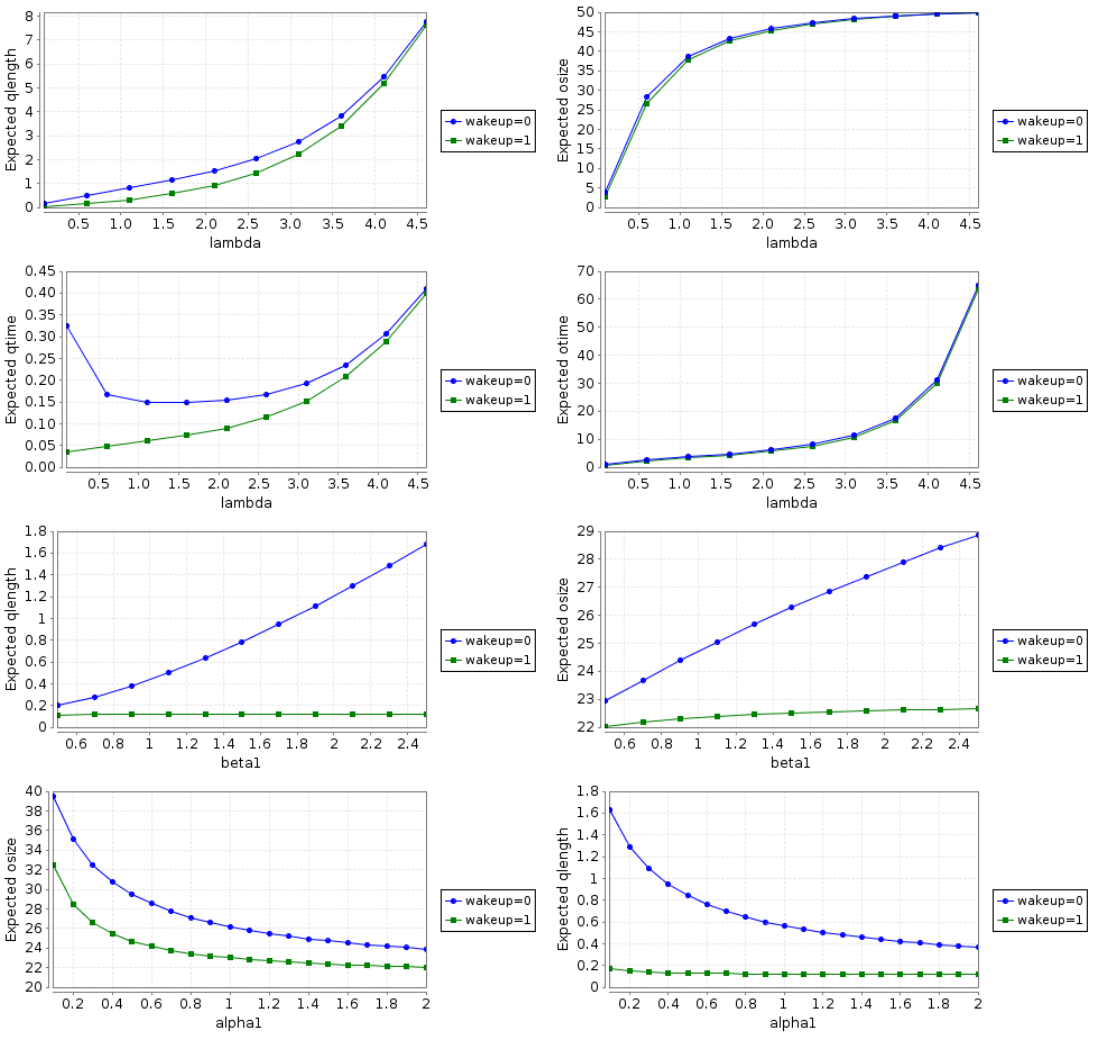

Figure 2: Finite Source: Measures ($\lambda = 0.5, \alpha_1 = 1/\alpha = 1.5, \beta_1 = 1/\beta = 1.0$)

- the bottom 2 diagrams of Figure 2 show the same shape but different absolute values as the curves given in Figures 12 and 13 of [3].

As for the last two diagrams, a little inspection of the corresponding figures in [3] reveals that they are not consistent with the values given there in Figures 3 and 4; apparently for their production erroneously different numerical parameters were used.

3 Infinite Source Model

In this section, we present an alternative model by lifting for normal sensors the restriction that they cannot trigger new data transmission requests before their previous requests have been satisfied. The model thus turns (for normal requests) from a "finite source" queueing model to an "infinite source" model. The differences of the new model (which is listed in Appendix B) to the finite source model are the following:

```
const int B = 250; // size of orbit

module Normal
  [nserver] true -> K*lambda2 : true;
  [norbit]  true -> K*lambda2 : true;
endmodule

module Orbit
  o: [0..B] init 0;
  [norbit]  o < B & !(ison = true & job = 0 & q = 0) -> (o' = o+1);
  [oserver] o > 0 -> o*nu : (o' = o-1);
endmodule
```

Module *Normal* does not capture any more the number of active sensors; instead it produces requests at the fixed rate $K \cdot \lambda_2$ which are directed (depending on the state of the RF unit) either to the orbit or to the RF unit. Thus the also the synchronized action *ndone* is removed from the module. Since the number of transmission requests in the orbit is not bounded any more by the number K of normal sensors, we introduce a parameter B for the maximum size of the orbit. By an additional reward structure

```
rewards "reject"
  o = B : 1;
endrewards
```

we check the probability that the orbit becomes full, i.e., that new transmission requests from normal sensors are "rejected".

The top diagram in Figure 3 lists this probability[2]. We see that for $\lambda \geq 0.4$, the probability of rejection becomes substantial; in the following we thus restrict our considerations to the range $\lambda \leq 0.6$. Figures 3, 4, and 5 depict in the left column of diagram probabilities and measures for the finite source model presented in the previous section while the right column depicts the corresponding infinite source model.

[2]Here and in the following pictures we use the default parameter values $\lambda = 0.3, \alpha_1 = 1/\alpha = 1.5, \beta_1 = 1/\beta = 1.0$.

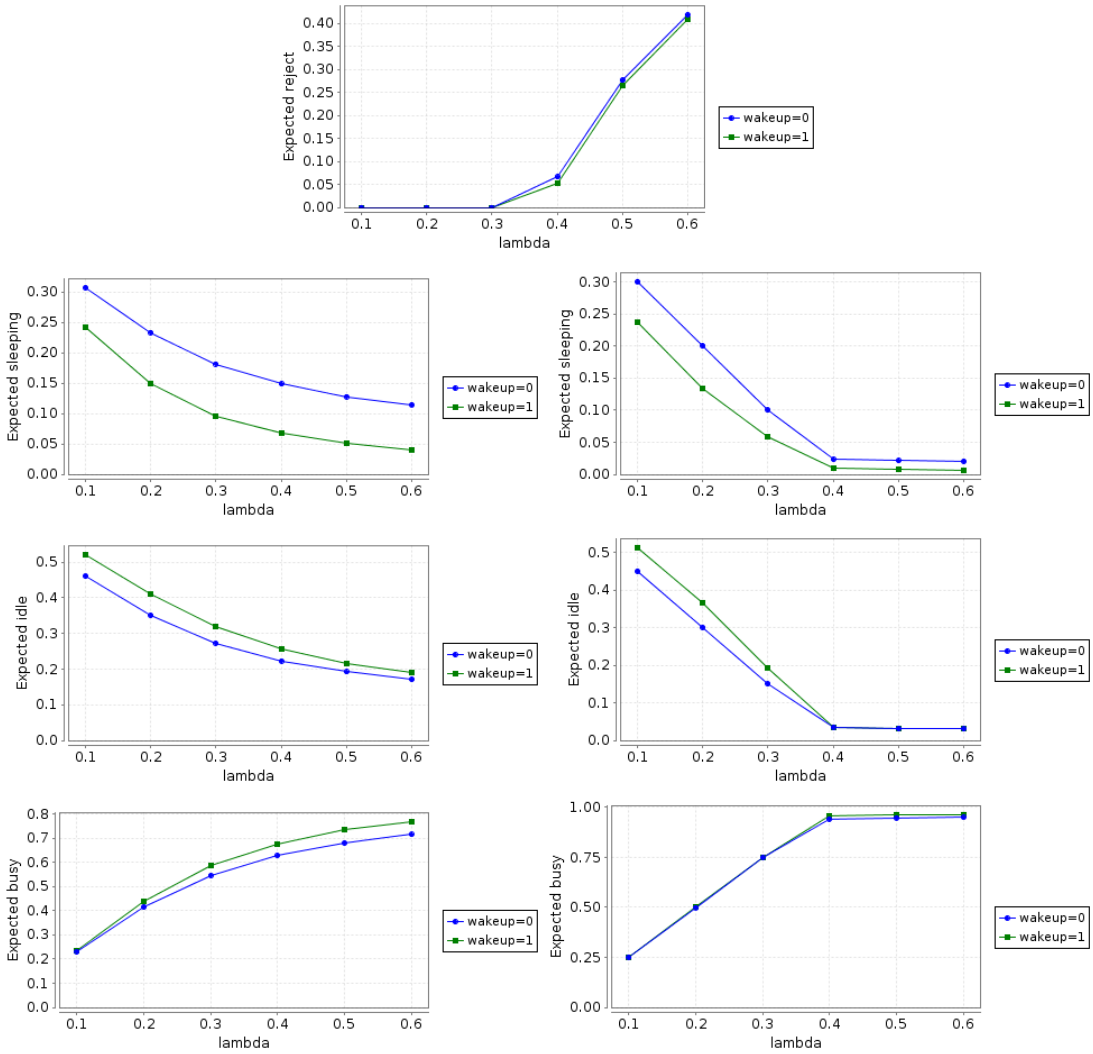

Figure 3: Finite/Infinite Source: Probabilities ($\lambda = 0.3, \alpha_1 = 1/\alpha = 1.5, \beta_1 = 1/\beta = 1.0$)

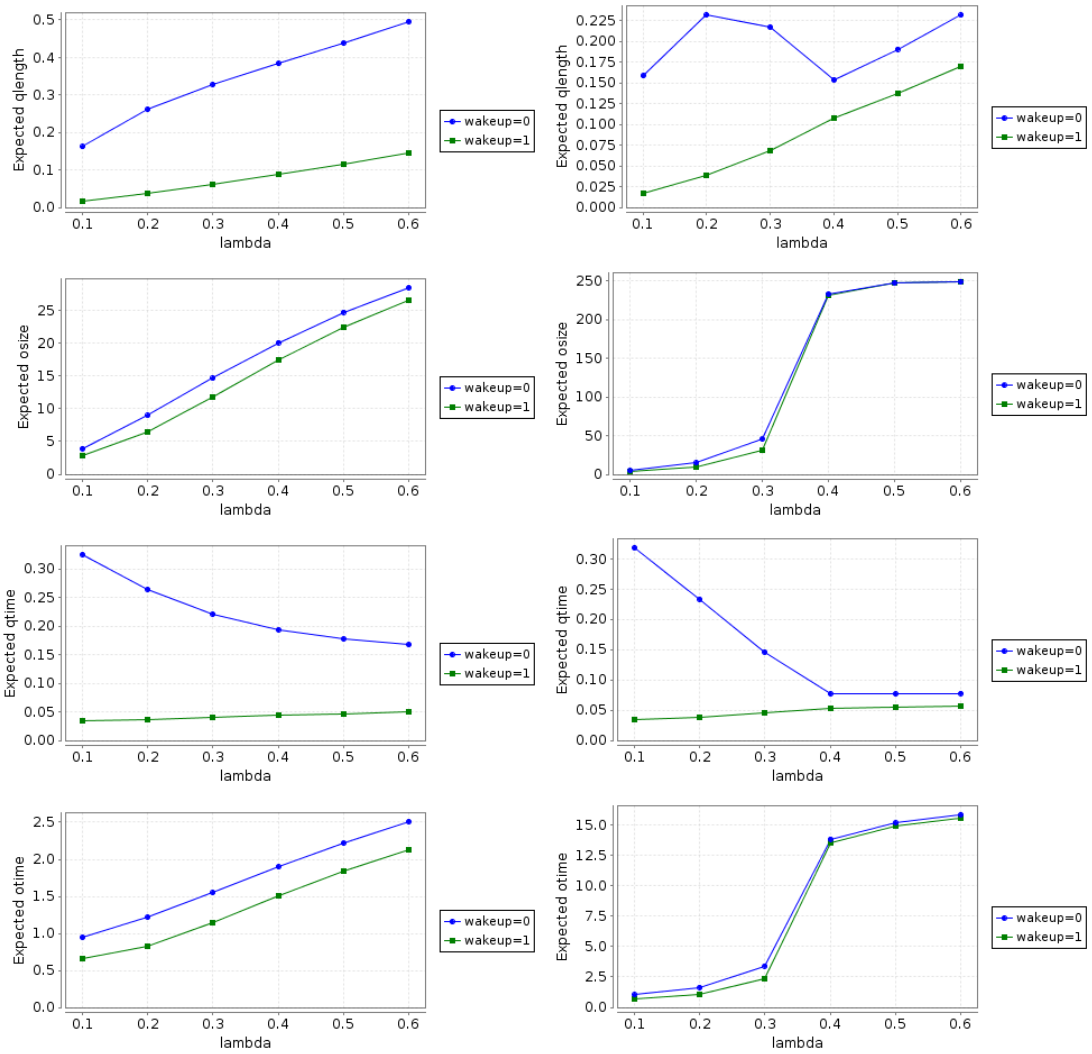

Figure 4: Finite/Infinite Source: Measures ($\lambda = 0.3, \alpha_1 = 1/\alpha = 1.5, \beta_1 = 1/\beta = 1.0$)

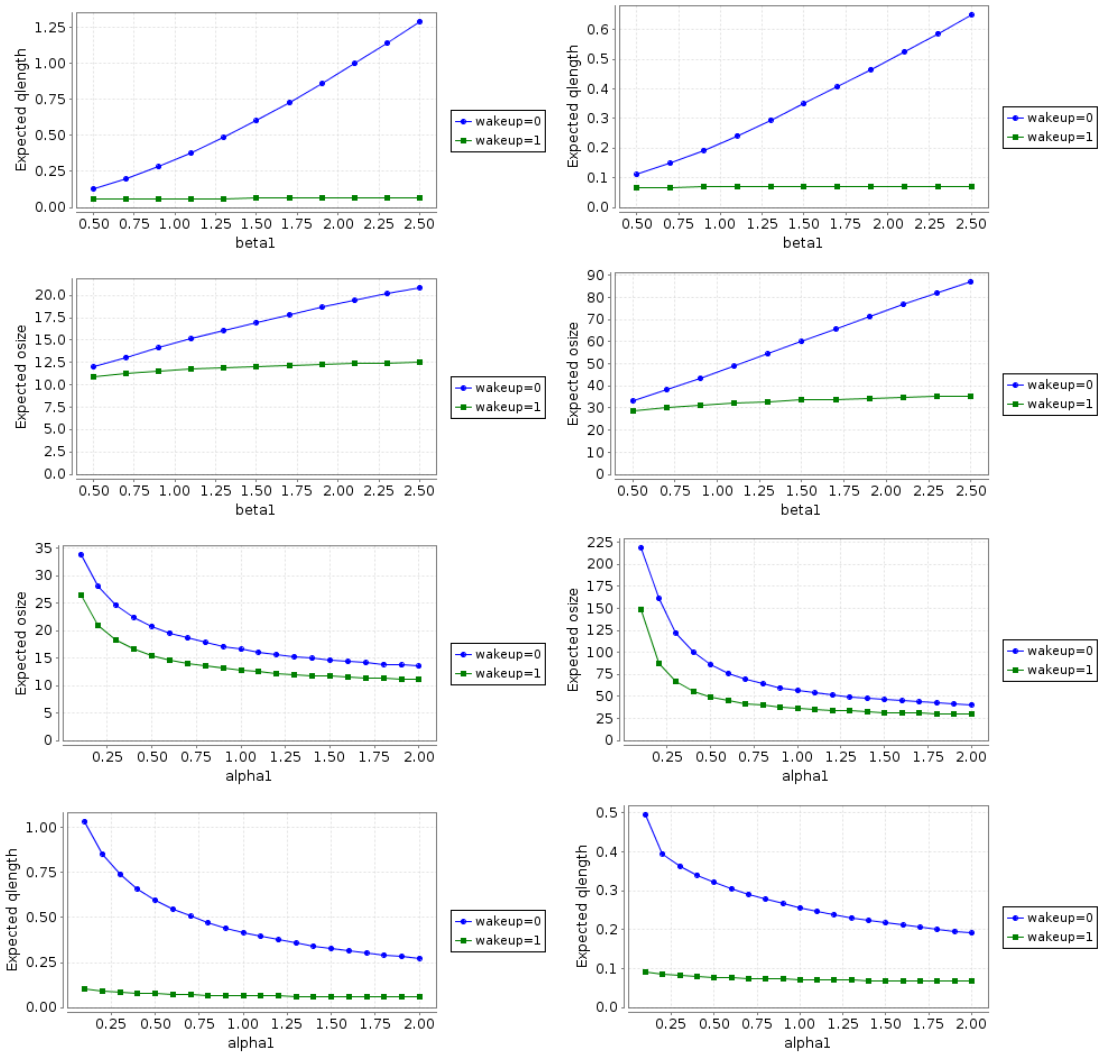

Figure 5: Finite/Infinite Source: Measures ($\lambda = 0.3, \alpha_1 = 1/\alpha = 1.5, \beta_1 = 1/\beta = 1.0$)

- We see that the probability of the RF unit being "busy" also for small values of λ quickly approaches one. This is caused by the substantially increased rate at which transmission requests from normal sensors arrive preventing the unit from sleeping or getting idle.
- Likewise, we see that the number of elements in the orbit and correspondingly the time of requests remaining in the orbit get quickly very large and reach for $\lambda \geq 0.4$ almost their maxima.
- However, the effect for the number of elements in the queue is different; while there is essentially no change in the mode with "wakeup", in the mode without this optimization, emergency requests are processed much quicker than before; this is due to the increased rate of normal requests which prevents the RF units from switching to the "off" state.

In the experiments up to now, the arrival rates λ_1 of emergency requests and λ_2 of normal requests where in a constant proportion $1 : 9$. Since the effective rate of normal requests, however, is now effectively increased, we fix now the value $\lambda_1 = 0.1$ and let vary λ_2 independently between 0.1 and 0.35. The results are illustrated in Figures 6 and 7:

- The top diagram in Figure 6 shows that the rejection rate is insignificant for $\lambda_2 \leq 0.25$, tolerable for $\lambda_2 = 0.3$ and becomes significant for $\lambda_2 \geq 0.35$.
- Comparing in Figure 6 the probabilities in the left column with the finite source model to the probabilities in the right column with the infinite source model, we see that the effect of the "wakeup" optimization becomes smaller for growing λ_2 and insignificant for $\lambda_2 \geq 0.3$.
- Analogously the results in Figure 7 show the vanishing advantage of the "wakeup" optimization for $\lambda_2 \geq 0.3$.

All in all, the "wakeup" optimization turns out to be of value only for a quite small range for the rate λ_2 of transmission requests from normal sensors; as soon as this value reaches a critical threshold, its advantage vanishes.

4 Conclusions

In this paper, we have validated with the help of the probabilistic model checker PRISM some previously reported results on RF communication in sensor networks and the effect of a proposed optimization of "emergency wakeups" of the RF unit to the responsiveness of the system (also exhibiting an error in a previous paper). This result again demonstrates that probabilistic model checking is on par with other approaches to performance modeling and analysis; moreover the PRISM models are very transparent and make hidden design decisions quite explicit.

Furthermore, we have used this opportunity to extend the previously reported and analyzed model to consider some variation of the model where the generation of new transmission requests of sensors is not coupled to the finishing of previous requests. It is shown that in this model the "emergency wakeup" optimization only is beneficial for a low rate of normal sensor transmissions. In the future, we plan to extend the analysis to other variations of RF communication models as well.

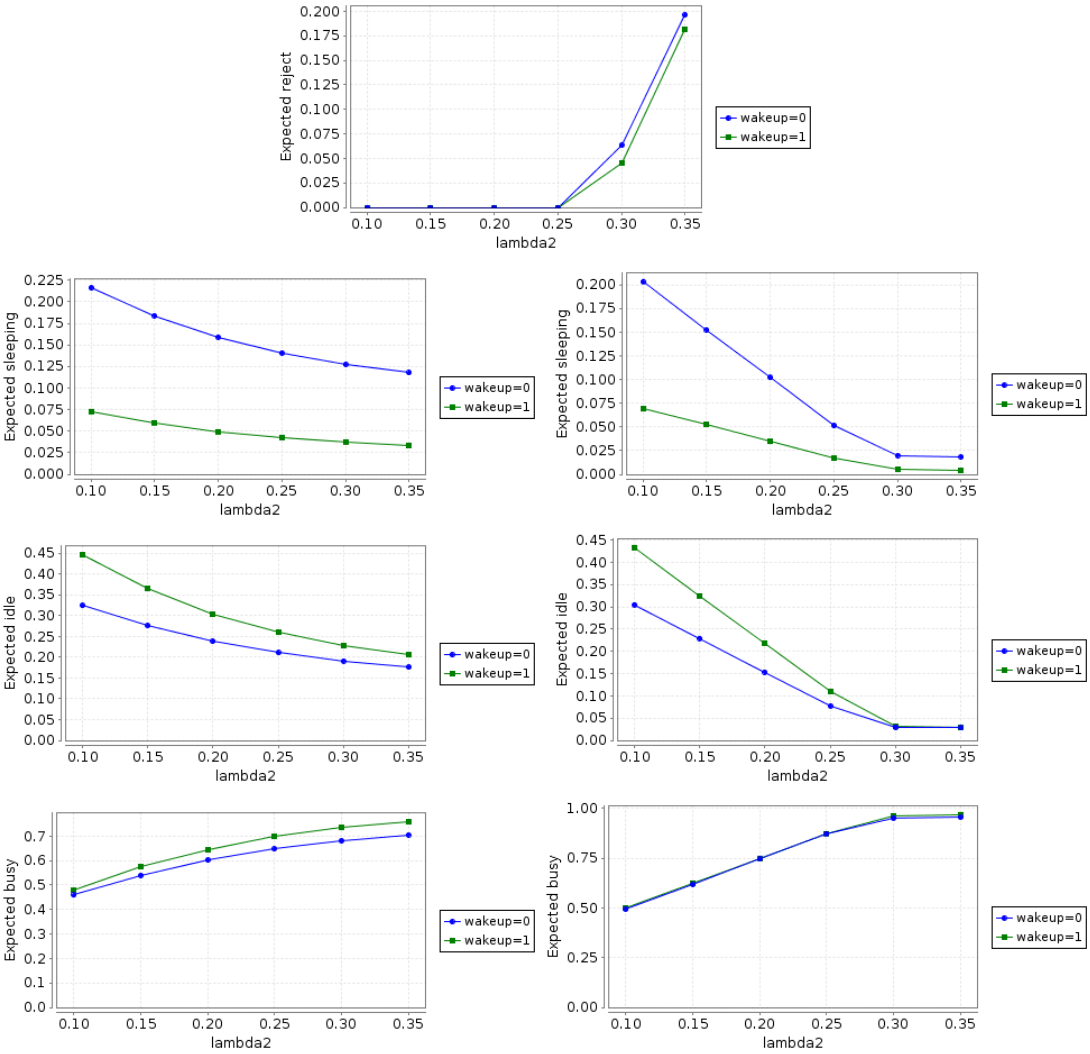

Figure 6: Finite/Infinite Source: Probabilities ($\lambda_1 = 0.1, \alpha_1 = 1/\alpha = 1.5, \beta_1 = 1/\beta = 1.0$)

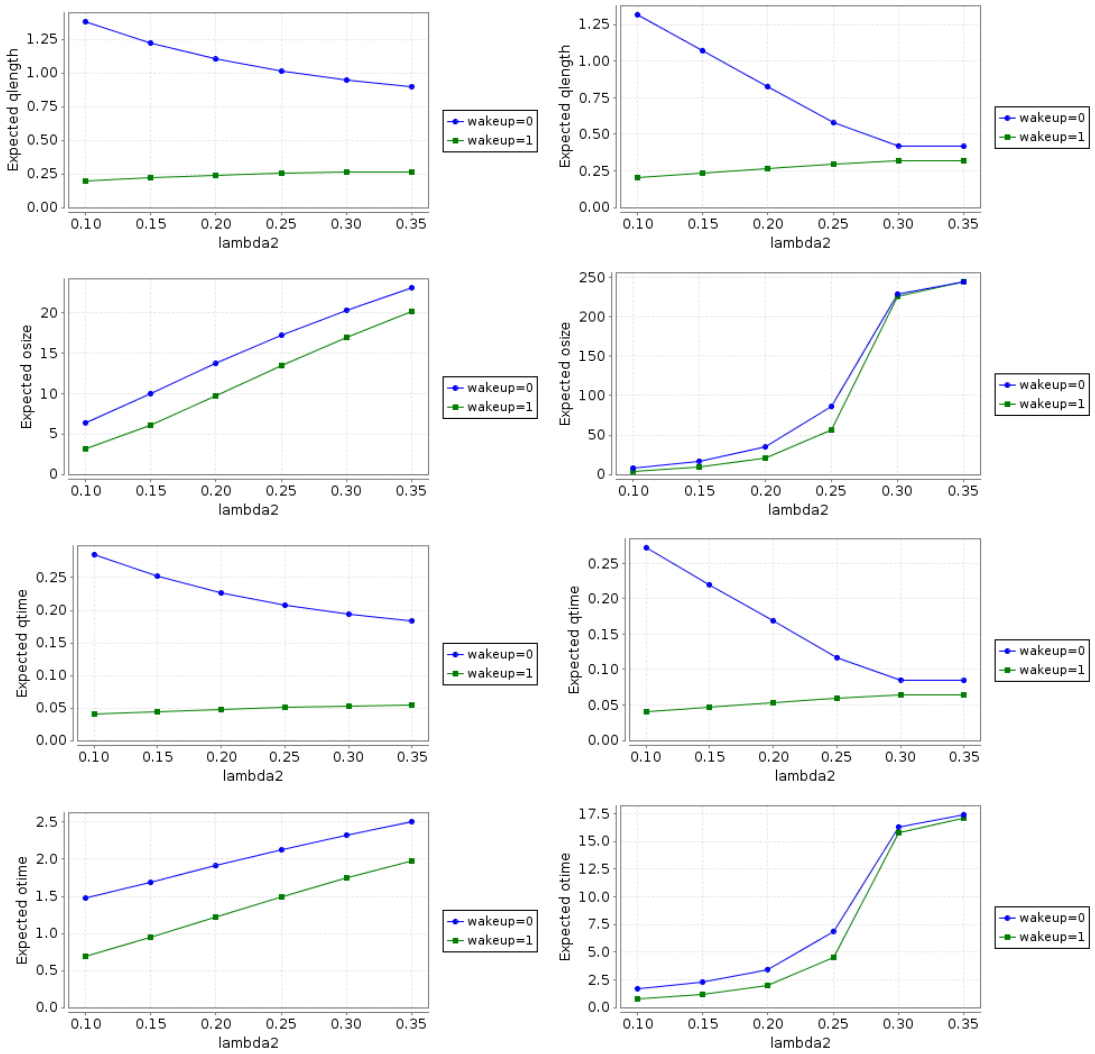

Figure 7: Finite/Infinite Source: Measures ($\lambda_1 = 0.1, \alpha_1 = 1/\alpha = 1.5, \beta_1 = 1/\beta = 1.0$)

References

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

[2] Tamás Bérczes, Béla Almási, János Sztrik, and Attila Kuki. "A Contribution to Modeling Sensor Communication Networks by Using Finite-Source Queueing Systems". In: *8th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*. Timisoara, Romania, May 23–25, 2013, pp. 89–93.

[3] Tamás Bérczes, Béla Almási, János Sztrik, and Attila Kuki. "Modeling the RF Communication in Sensor Networks by using Finite-Source Retrial Queueing System". In: *Scientific Bulletin of the Politehnica University of Timisoara, Romania, Transactions on Automatic Control and Computer Science* 58(72).2–4 (Mar. 2013).

[4] 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 (C AV'11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 585–591.

[5] David A. Parker, ed. *PRISM — Probabilistic Symbolic Model Checker*. http://www.prismmodelchecker.org. Department of Computer Science, University of Oxford, UK. 2015.

A Finite Source Model

A.1 The Original Model

```
// ------------------------------------------------------------------
// Finite.prism
// A model for RF communication in sensor networks.
//
// The model is described in
//
//   Tamas Berczes, Bela Almasi, Janos Sztrik and Attila Kuki.
//   Modeling the RF Communication in Sensor Networks by using
//   Finite-Source Retrial Queueing System.
//   Scientific Bulletin of the Politehnica University of Timisoara, Romania,
//   Transactions on Automatic Control and Computer Science, Vol. 58(72),
//   No. 2-4, Mach-December 2013.
//
// Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at> and
// Berczes Tamas <berczes.tamas@inf.unideb.hu>
//
// Copyright (C) 2015 Research Institute for Symbolic Computation,
// Johannes Kepler University, Linz, Austria (http://www.risc.jku.at) and
// Department of Informatics Systems and Networks, University of Debrecen,
// Debrecen, Hungary (http://irh.inf.unideb.hu)
// ------------------------------------------------------------------

// checking parameters: "sparse", "Jacobi", epsilon=10^-6

// continuous time markov chain (ctmc) model
ctmc

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

const int N = 50;  // number of emergency sensors
const int K = 50;  // number of standard sensors

const double lambda;               // overall generation rate [0.1,4.6]
const double lambda1 = lambda*0.1; // emergency generation rate
const double lambda2 = lambda*0.9; // standard generation rate

const double nu = 2;     // retrial rate
const double mu = 20;    // service rate
const double gamma = 10; // initialization rate

const double alpha1; // =1/alpha: mean time of listening period [0.1,2] or 1.5
const double beta1;  // =1/beta:  mean time of sleeping period [0.5,2.5]

const double infinity = 9999; // an "infinite rate"

const int wakeup; // 1: emergency request wakes up RF unit

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

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

module Normal
  k2: [0..K] init K;
  [nserver] k2 > 0 -> k2*lambda2 : (k2' = k2-1);
  [norbit]  k2 > 0 -> k2*lambda2 : (k2' = k2-1);
  [ndone]   k2 < K -> (k2' = k2+1);
endmodule

module Orbit
  o: [0..K] init 0;
  [norbit]  o < K & !(ison = true & job = 0 & q = 0) -> (o' = o+1);
  [oserver] o > 0 -> o*nu : (o' = o-1);
endmodule

module Emergency
  k1: [0..N] init N;
  [equeue] k1 > 0 -> k1*lambda1 : (k1' = k1-1);
  [edone]  k1 < N -> (k1' = k1+1);
endmodule

module Queue
  q: [0..N] init 0;
  [equeue]  q < N -> (q' = q+1);
  [qserver] q > 0 -> (q' = q-1);
endmodule

module Server
  ison: bool init true;
  job: [0..2] init 0; // 1: normal job, 2: emergency job
  [switchoff] ison = true & job = 0 & q = 0 -> 1/alpha1 : (ison' = false);
  [switchon]  ison = false -> 1/beta1 : (ison' = true);
  [wakeup]    ison = false & q > 0 & wakeup = 1 -> gamma : (ison' = true);
  [nserver]   ison = true & job = 0 & q = 0 -> (job' = 1);
  [oserver]   ison = true & job = 0 & q = 0 -> (job' = 1);
  [ndone]     job = 1 -> mu : (job' = 0);
  [qserver]   ison = true & job = 0 -> infinity : (job' = 2);
  [edone]     job = 2 -> mu : (job' = 0);
endmodule

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

rewards "qlength"
  true : q;
endrewards

rewards "osize"
  true : o;
endrewards

rewards "eactive"
```

```
  true : k1;
endrewards

rewards "nactive"
  true : k2;
endrewards

rewards "sleeping"
  !ison : 1;
endrewards

rewards "idle"
  ison & job = 0: 1;
endrewards

rewards "busy"
  ison & job != 0: 1;
endrewards

// ------------------------------------------------------------------
// end of file
// ------------------------------------------------------------------
```

A.2 The Streamlined Model

```
// ------------------------------------------------------------------
// Finite.prism
// A model for RF communication in sensor networks.
//
// The model is described in
//
//   Tamas Berczes, Bela Almasi, Janos Sztrik and Attila Kuki.
//   Modeling the RF Communication in Sensor Networks by using
//   Finite-Source Retrial Queueing System.
//   Scientific Bulletin of the Politehnica University of Timisoara, Romania,
//   Transactions on Automatic Control and Computer Science, Vol. 58(72),
//   No. 2-4, Mach-December 2013.
//
// Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at> and
// Berczes Tamas <berczes.tamas@inf.unideb.hu>
//
// Copyright (C) 2015 Research Institute for Symbolic Computation,
// Johannes Kepler University, Linz, Austria (http://www.risc.jku.at) and
// Department of Informatics Systems and Networks, University of Debrecen,
// Debrecen, Hungary (http://irh.inf.unideb.hu)
// ------------------------------------------------------------------

// checking parameters: "sparse", "Jacobi", epsilon=10^-6

// continuous time markov chain (ctmc) model
ctmc

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

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

const int N = 50;  // number of emergency sensors
const int K = 50;  // number of standard sensors

const double lambda;               // overall generation rate [0.1,4.6]
const double lambda1 = lambda*0.1; // emergency generation rate
const double lambda2 = lambda*0.9; // standard generation rate

const double nu = 2;     // retrial rate
const double mu = 20;    // service rate
const double gamma = 10; // initialization rate

const double alpha1; // =1/alpha: mean time of listening period [0.1,2] or 1.5
const double beta1;  // =1/beta:  mean time of sleeping period [0.5,2.5]

const int wakeup; // 1: emergency request wakes up RF unit

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

module Normal
  k2: [0..K] init K;
  [nserver] k2 > 0 -> k2*lambda2 : (k2' = k2-1);
  [norbit]  k2 > 0 -> k2*lambda2 : (k2' = k2-1);
  [ndone]   k2 < K -> (k2' = k2+1);
endmodule

module Orbit
  o: [0..K] init 0;
  [norbit]  o < K & !(ison = true & job = 0 & q = 0) -> (o' = o+1);
  [oserver] o > 0 -> o*nu : (o' = o-1);
endmodule

module Emergency
  k1: [0..N] init N;
  [equeue] k1 > 0 -> k1*lambda1 : (k1' = k1-1);
  [edone]  k1 < N -> (k1' = k1+1);
endmodule

module Queue
  q: [0..N] init 0;
  [equeue] q < N -> (q' = q+1);
  [edone]  q > 0 -> (q' = q-1);
endmodule

module Server
  ison: bool init true;
  job: [0..1] init 0; // 1: normal job
  [switchoff] ison = true & job = 0 & q = 0 -> 1/alpha1 : (ison' = false);
  [switchon]  ison = false -> 1/beta1 : (ison' = true);
  [wakeup]    ison = false & q > 0 & wakeup = 1 -> gamma : (ison' = true);
```

```
  [nserver]   ison = true & job = 0 & q = 0 -> (job' = 1);
  [oserver]   ison = true & job = 0 & q = 0 -> (job' = 1);
  [ndone]     job = 1 -> mu : (job' = 0);
  [edone]     ison = true & job = 0 -> mu : true;
endmodule

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

rewards "qlength"
    ison & job = 0 & q > 0  : q-1;
  !(ison & job = 0 & q > 0) : q;
endrewards

rewards "osize"
  true : o;
endrewards

rewards "eactive"
  true : k1;
endrewards

rewards "nactive"
  true : k2;
endrewards

rewards "sleeping"
  !ison : 1;
endrewards

rewards "idle"
  ison & job = 0 & q = 0: 1;
endrewards

rewards "busy"
  ison & !(job = 0 & q = 0): 1;
endrewards

// ---------------------------------------------------------------------
// end of file
// ---------------------------------------------------------------------
```

A.3 The CSL Queries

```
"qlength":  R{"qlength"}=? [ S ]
"osize":    R{"osize"}=? [ S ]
"eactive":  R{"eactive"}=? [ S ]
"nactive":  R{"nactive"}=? [ S ]
"qtime":    "qlength"/(lambda1*"eactive");
"otime":    "osize"/(lambda2*"nactive");
"sleeping": R{"sleeping"}=? [ S ]
"idle":     R{"idle"}=? [ S ]
```

```
"busy":     R{"busy"}=? [ S ]
```

B Infinite Source Model

B.1 The Model

```
// -------------------------------------------------------------------
// Infinite.prism
// A model for RF communication in sensor networks.
//
// The model is derived from the finite source model described in
//
//   Tamas Berczes, Bela Almasi, Janos Sztrik and Attila Kuki.
//   Modeling the RF Communication in Sensor Networks by using
//   Finite-Source Retrial Queueing System.
//   Scientific Bulletin of the Politehnica University of Timisoara, Romania,
//   Transactions on Automatic Control and Computer Science, Vol. 58(72),
//   No. 2-4, Mach-December 2013.
//
// Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at> and
// Berczes Tamas <berczes.tamas@inf.unideb.hu>
//
// Copyright (C) 2015 Research Institute for Symbolic Computation,
// Johannes Kepler University, Linz, Austria (http://www.risc.jku.at) and
// Department of Informatics Systems and Networks, University of Debrecen,
// Debrecen, Hungary (http://irh.inf.unideb.hu)
// -------------------------------------------------------------------

// checking parameters: "sparse", "Gauss-Seidel", epsilon=10^-6

// continuous time markov chain (ctmc) model
ctmc

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

const int N = 50;  // number of emergency sensors
const int K = 50;  // number of standard sensors
const int B = 250; // size of orbit

const double lambda;               // overall generation rate [0.1,4.6]
const double lambda1 = lambda*0.1; // emergency generation rate [0.01,0.46]
const double lambda2 = lambda*0.9; // standard generation rate [0.09, 4.14]

const double nu = 2;     // retrial rate
const double mu = 20;    // service rate
const double gamma = 10; // initialization rate

const double alpha1; // =1/alpha: mean time of listening period [0.1,2] or 1.5
const double beta1;  // =1/beta:  mean time of sleeping period [0.5,2.5]

const int wakeup; // 1: emergency request wakes up RF unit
```

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

module Normal
  [nserver] true -> K*lambda2 : true;
  [norbit]  true -> K*lambda2 : true;
endmodule

module Orbit
  o: [0..B] init 0;
  [norbit]  o < B & !(ison = true & job = 0 & q = 0) -> (o' = o+1);
  [oserver] o > 0 -> o*nu : (o' = o-1);
endmodule

module Emergency
  k1: [0..N] init N;
  [equeue] k1 > 0 -> k1*lambda1 : (k1' = k1-1);
  [edone]  k1 < N -> (k1' = k1+1);
endmodule

module Queue
  q: [0..N] init 0;
  [equeue] q < N -> (q' = q+1);
  [edone]  q > 0 -> (q' = q-1);
endmodule

module Server
  ison: bool init true;
  job: [0..1] init 0; // 1: normal job
  [switchoff] ison = true & job = 0 & q = 0 -> 1/alpha1 : (ison' = false);
  [switchon]  ison = false -> 1/beta1 : (ison' = true);
  [wakeup]    ison = false & q > 0 & wakeup = 1 -> gamma : (ison' = true);
  [nserver]   ison = true & job = 0 & q = 0 -> (job' = 1);
  [oserver]   ison = true & job = 0 & q = 0 -> (job' = 1);
  [ndone]     job = 1 -> mu : (job' = 0);
  [edone]     ison = true & job = 0 -> mu : true;
endmodule

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

rewards "qlength"
    ison & job = 0 & q > 0  : q-1;
  !(ison & job = 0 & q > 0) : q;
endrewards

rewards "osize"
  true : o;
endrewards

rewards "eactive"
```

```
  true : k1;
endrewards

rewards "reject"
  o = B : 1;
endrewards

rewards "sleeping"
  !ison : 1;
endrewards

rewards "idle"
  ison & job = 0 & q = 0: 1;
endrewards

rewards "busy"
  ison & !(job = 0 & q = 0): 1;
endrewards

// ------------------------------------------------------------------
// end of file
// ------------------------------------------------------------------
```

B.2 The CSL Queries

```
"qlength":  R{"qlength"}=? [ S ]
"osize":    R{"osize"}=? [ S ]
"eactive":  R{"eactive"}=? [ S ]
"reject":   R{"reject"}=? [ S ]
"qtime":    "qlength"/(lambda1*"eactive");
"otime":    "osize"/(lambda2*K*(1-"reject"));
"sleeping": R{"sleeping"}=? [ S ]
"idle":     R{"idle"}=? [ S ]
"busy":     R{"busy"}=? [ S ]
```