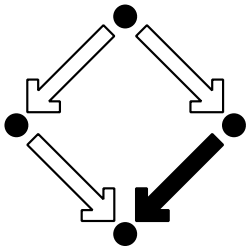
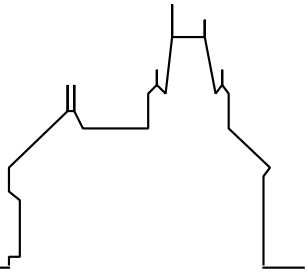


RISC-Linz

Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe



On the Probabilistic Model Checking of a Retrial Queueing System with Unreliable Server, Collision, and Constant Time Impatience

Wolfgang SCHREINER and János SZTRIK

(July 2019)

RISC-Linz Report Series No. 19-11

Supported by: Austrian-Hungarian Bilateral Cooperation in Science and Technology project 2017-2.2.4-TeT-AT-2017-00010 and the Aktion sterreich-Ungarn project 101u7.

On the Probabilistic Model Checking of a Retrial Queueing System with Unreliable Server, Collision, and Constant Time Impatience*

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

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

July 16, 2019

Abstract

We report on initial experiments with the automated analysis of a finite-source queueing system with an unreliable server and collisions of service requests that cause clients to be moved to an orbit until they can be served. However, clients remain in the obit only for some maximum amount of time before they run out of patience and unsuccessfully abort their service request. In contrast to earlier investigations, the duration of their patience is not exponentially distributed but constrained by a constant time bound, which imposes a problem for both their manual and automatic analysis. In this paper we address how such systems can be nevertheless approximately analyzed to a certain extent with the help of the probabilistic model checker PRISM.

*Supported by the Austrian-Hungarian Bilateral Cooperation in Science and Technology project 2017-2.2.4-TeT-AT-2017-00010 and the Aktion Österreich-Ungarn project 101öu7.

Contents

1 Introduction

This paper presents some initial work on the automatic analysis of retrial queueing systems with impatience [10] where service requests may be prematurely aborted, if customers run out of patience when waiting to be served. Typically the duration of patience has been assumed to be exponentially distributed, which allows the manual and automatic analysis of corresponding systems. In contrast, for patience with constant time bounds, it seems that only simulation techniques are applicable, as we have also used in other contexts [6]. The goal of this work is to investigate how far also tools not based on simulation, such as the probabilistic model checker PRISM [5, 7], can be applied for that purpose.

In various previous work (see e.g. [9]), we have successfully applied PRISM to the performance modeling of various kinds of retrial queueing systems on the basis of Continuous Time Markov Chains (CTMCs) where transition times are exponentially distributed. However, there have also been some recent third-party extensions to PRISM that deal with other types of distributions:

- fdPRISM [3, 1] is an experimental extension of PRISM that supports "fixed delays", exactly what we need. Unfortunately, however, this extension only supports a very restricted form of analysis, where the accumulated reward of a system is computed until a certain target state is reached. This kind of analysis is too rigid for our goals; also since 2016 the development of the system seems to be frozen.

- PRISM GSMP [2, 8] extends PRISM by Generalized Semi-Markov Processes (GSMPs) which may be seen as CTMCs with any number of concurrently active state-changing "events" at any given time. These events may follow various kinds of distributions, in particular also the deterministic "Dirac" distribution which takes a single "timeout" value $t > 0$. While this is indeed exactly what we need, the PRISM GSMP software actually only supports the analysis of "Continuous Time Markov Chains with Alarms" (ACTMCs) where there is at most *one* non-exponentially distributed event active in every given state. Since our scenario involves multiple customers each of which has a separate "patience timeout", this restriction is too strong for our purpose.

Since therefore none of these extensions is thus applicable for the scenarios we envision, we have to investigate how else we can deal in PRISM with constant time bounds. Here the PRISM FAQ [7] gives in the question "How can I add deterministic time delays to a CTMC model?" the following crucial advice:

> All delays in a CTMC need to be modeled as exponential distributions. This is what makes them efficient to analyze. If you included a transition whose delay was deterministic, i.e. which always occurred after exactly the same delay, the model would no longer be a CTMC.
>
> One solution to this, if your model require such a delay, is to approximate a deterministic delay with an Erlang distribution (a special case of a phase-type distribution).
> ...

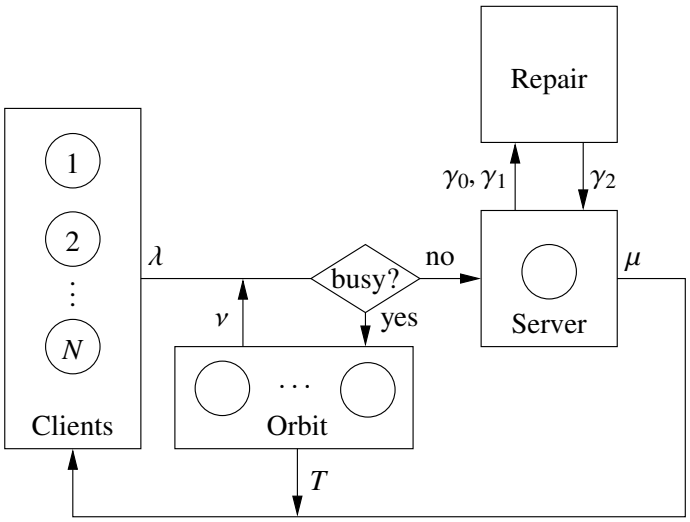


Figure 1: The System Model

In a nutshell, the advice is to equip the model with a particular "alarm clock", an automaton that triggers an action after a sequence of $k > 0$ transitions each of which is exponentially distributed with mean time t/k; consequently the time after which the alarm "rings" follows an Erlang distribution with mean time t and shape k. The special case $k = 1$ represents the exponential distribution; the larger k is, the more "deterministic" the time variable has value t. The trade-off, however, is that such an alarm clock causes a "blow-up" of the the size of the model by a factor k, with correspondingly harsh consequences on the time and space needed for analyzing the model.

In this paper, we will investigate how far it makes sense to follow this advice for the scenario we have in mind. The rest of the paper is structured as follows: in Section 2 we describe in detail our abstract system model and its concrete realization in PRISM; in Section 3, we show the results of some initial experiments with analyzing the model; Section 4 presents our preliminary conclusions. Appendix A gives the full definitions of the PRISM model and of the queries used to analyze this model.

2 The System Model

We consider a system with the following characteristics (see Figure 1), a version of the system presented in [4] generalized to consider the impatience of clients:

- There are N clients that request being served at rate λ.
- A server processes client requests at rate μ; after a client has been served, it may start another service request.

- If a client finds upon a newly generated request the server busy with serving another client, a *collision* occurs and both clients move to an *orbit*.
- A client in the orbit retries its service request at rate ν; also a retrial may cause a collision in the server, which moves the currently served client into the orbit.
- If a client in the orbit does not find the server ready to accept its service request after T time units, it *aborts* the current request and becomes ready to start another one.
- If the server is idle, it may *fail* with rate γ_0. If the server is busy, it may fail with rate γ_1; this moves the currently served client to the orbit.
- While being in the failure state, the server does not serve clients; and all newly arriving requests go into the orbit. A failed server is repaired with rate γ_2 and then becomes ready to serve clients again.

The parameters of the model are therefore as follows:

Parameter	Interpretation
N	number of clients
λ	rate of service requests (arrival rate)
μ	rate of service completions (departure rate)
ν	rate of service retrials (retrial rate)
T	maximum duration of service retrial (patience duration)
γ_0	failure rate in idle state
γ_1	failure rate in busy state
γ_2	repair rate

Here all rates denote the parameters of exponential distributions; however, T is a constant (deterministic) time.

In PRISM, we can approximate constant time T by an Erlang distribution with shape K as generated by the following model (adapted from the example in the FAQ section of [7]):

```
ctmc
const double T = 10;
const int K;
module alarm
  t: [0..K] init 1;
  a: [0..1] init 0
  [tick]  t < K -> K/T : (t' = t+1);
  [alarm] t = K -> K/T : (a' = 1);
endmodule
```

Analyzing this model with the CSL query

```
const double T0;
P=? [ F<=T0 a=1 ]
```

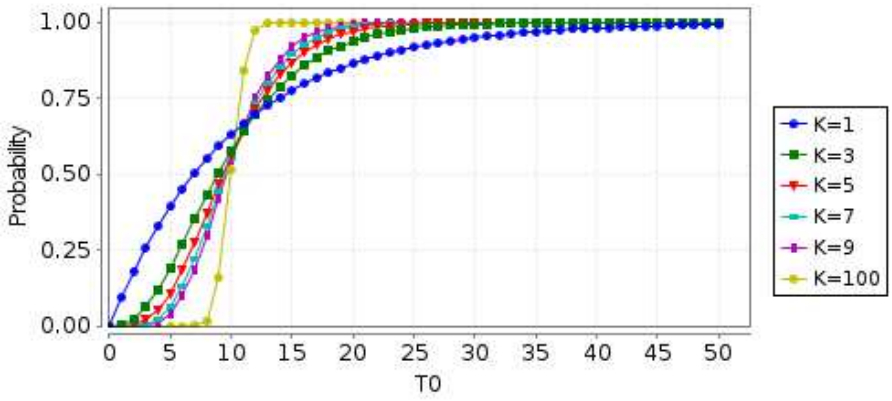


Figure 2: Erlang Distribution

we can determine the probability that the transition *alarm* is fired at time T_0. As Figure 2 demonstrates, the more K grows, the more "deterministically" T_0 approximates the model constant $T = 10$. The case $K = 1$ represents the exponential distribution.

Using these ideas, Appendix A now gives the specification of a PRISM model for the system sketched above. In essence, this model contains a server module and N client modules where server and clients are synchronized by shared transitions.

The server module can be in one of four states indicated below:

```
module Server
  s: [0..3] init 0; // 0: idle, 1: busy, 2: collision, 3: failed

  // evict server due to collision
  [evict] s = 2 -> infinity : (s' = 0);

  // evict server due to failure and then repair it
  [evict]  s = 0 -> gamma0 : (s' = 3);
  [evict]  s = 1 -> gamma1 : (s' = 3);
  [repair] s = 3 -> gamma2 : (s' = 0);

  // interaction with client 1
  [source1] N >= 1 & s = 0 ->      (s' = 1);
  [orbit1]  N >= 1 & s = 0 ->      (s' = 1);
  [exit1]   N >= 1 & s = 1 -> mu : (s' = 0);
  [scoll1]  N >= 1 & s = 1 ->      (s' = 2);
  [ocoll1]  N >= 1 & s = 1 ->      (s' = 2);
  [scoll1]  N >= 1 & s = 3 ->      true;

  ...
endmodule
```

In the "idle" state $s = 0$ the server may accept by a transition *sourceC* a request from client C

in the source or by a transition *orbitC* a request from that client in the orbit, thus switching to the "busy" state $s = 1$. In that state, the server may by transition *exitC* successfully process that request or experience by transitions *scollC* or *ocollC* collisions with requests from other clients in the source or in the orbit, thus switching to the "collision" state $s = 2$. In that state, an "infinity rate" transition $s = 2$ is possible that will subsequently cause by a transition *evict* the "eviction" of the currently served client from the server (see below); also a transition *scoll1* is possible that corresponds to the failed attempt of a new request to enter the server. Evictions also occur when the server fails (in either the idle or the busy state), by which the server switches to the "failure" state $s = 3$. The server remains in that state in which it does not accept any requests until it is repaired and then becomes idle again, thus accepting new requests.

Every client is an instance of the following module for Client 1:

```
module Client1
  c1: [0..2] init 0; // 0: source, 1: server, 2: orbit
  t1: [0..K] init 0; // tick counter when in orbit

  // interaction with server
  [source1] c1 = 0 -> lambda : (c1' = 1);
  [scoll1]  c1 = 0 -> lambda : (t1' = 1);
  [orbit1]  c1 = 2 -> nu     : (c1' = 1) & (t1' = 0);
  [ocoll1]  c1 = 2 -> nu     : true;
  [exit1]   c1 = 1 ->          (c1' = 0);

  // go to orbit due to collision or failure
  [evict] c1 = 0 & t1 > 0  -> (c1' = 2);
  [evict] c1 = 0 & t1 = 0  -> true;
  [evict] c1 = 1           -> (c1' = 2) & (t1' = 1);
  [evict] c1 = 2           -> true;

  // in orbit countdown to abortion
  [tick1]  c1 = 2 & t1 < K -> K/T : (t1' = t1+1);
  [abort1] c1 = 2 & t1 = K -> K/T : (c1' = 0) & (t1' = 0);
endmodule
```

When in the "source" state $c_1 = 0$, the client may generate a request, which causes either by transition *source1* is accepted by the idle server or by transition *scoll1* causes a collision there. In case of acceptance, the client switches to the "server" state 1, from which it expects to switch after the service by transition *exit1* back to the source state $c_1 = 0$. In the case of a collision, the client sets the timer t_1 to 1 which differentiates the client that caused the collision from all the other ones.

As shown above, when a server experiences a collision it performs a transition *evict* which is synchronized with *all* clients. The client I with $c_I = 0$ and $t_I > 0$ that caused the collsion moves from the source to the "orbit" state $s = 2$ as well as the client I with $c_I = 1$ that was just being served (also setting its timer t_I to 1).

When Client 1 is in the orbit state $c_1 = 2$, it attempts to enter the server by transition *orbit1* but may also cause by transition *ocoll1* a collision in the server. Simultaneously, however, by the transition *tick1* the "impatience" timer "ticks" incrementing the value t_1. When that timer

reaches the value K, the transition $\iota abort 1\jmath$ aborts the request and the client moves back to the source.

A critical aspect in above model is that each client indeed requires its own "alarm timer" which is appropriately set when the client moves to the orbit and which causes upon "ringing" to move the client out of the orbit again. Thus indeed it does not suffice to have a single module that just counts the number of clients that are in the source respectively in the orbit but every client requires its own state. Having N clients with S possible states thus yields a system with S^N states, i.e., the size of the system grows exponentially with the number of clients. This will be the major limit in the automatic analysis of the system by the PRISM model checker described in Section 3.

Finally, to perform certain kinds of such an analysis, the system is also equipped with two "transition rewards":

```
rewards "NumExit"
  [exit1]  true : 1;
  [exit2]  true : 1;
  ...
endrewards

rewards "NumAbort"
  [abort1]  true : 1;
  [abort2]  true : 1;
  ...
endrewards
```

The transition reward $E :=$ *NumExit* "counts" every successful completion of a request by the server while $A :=$ *NumAbort* counts every premature abortion due to a client running out of patience. The long term ratio $E/(E + A) = 1/(1 + (A/E))$ thus determines the probability that a client request is successfully completed.

3 Experimental Evaluation

In the following, we analyze the model introduced in Section 2 in PRISM. For this we mainly use the CSL query

```
 const int i;
"Pi": S=? [ i = min(c1,1)+min(c2,1)+min(c3,1)+min(c4,1)+min(c5,1)
              +min(c6,1)+min(c7,1)+min(c8,1)+min(c9,1)+min(c10,1) ];
```

which calculates the probability $P(i)$ that there are i clients not in the source, i.e., that they are either in the orbit or being served. We apply for this analysis the PRISM "Hybrid" engine with the "Jacobi" solver using the relative termination criterion $\epsilon = 0.01$ (which is much larger than the standard value $\epsilon = 10^{-6}$; this value however lets the computation time explode without really significantly improving the accuracy of the results).

This evaluation is partially motivated by the corresponding data given in [4]; however, they are not comparable because in that paper numerical calculations were performed for a system without impatience on the basis of a manually derived equation system for $N = 100$. In our

model with constant time patience, the calculation for a single data point already with $N = 10$ and $K = 3$ takes one to two minutes; a computation with larger model sizes is completely out of reach (in particular also due to memory limitations, for $N = 10$ and $K = 5$ the models need multiple GB of memory).

Subsequently we use therefore (variations of) the following set of parameters:

$$N = 6, \lambda = 0.01, \mu = 0.06, \nu = 0.05, \gamma_0 = 0.006, \gamma_1 = 0.006, \gamma_2 = .06,$$
$$T = 50, K = 5.$$

For these parameters the model has about $7 \cdot 10^5$ states and the computation of a data point takes only 1–2 seconds (for $K = 9$, the model already has about $7 \cdot 10^6$ states and the computation of a data point takes about 20 seconds).

Consequently we start our investigation by analyzing in Figure 3 the effect of the shape K of the Erlang distribution on the analysis of the model. The upper diagram determines the probability $P(i)$ of a client being in the system; the lower diagram determines by the CSL query

```
"NE": R{"NumExit"}=? [ S ] ;
"NA": R{"NumAbort"}=? [ S ] ;
"PE": 1/(1+("NA"/"NE"));
```

the probability PE that a client request is not aborted due to impatience, i.e., that it is successfully served. While for $K < 5$ the results show significant differences, they are indeed quite close for $K \geq 5$. This justifies the use of the manageable value $K = 5$ in the remaining experiments, which reasonably well approximates a constant time (see Figure 2).

In Figure 4, we investigate the effect of varying arrival rate λ, departure rate μ, and retrial rate ν. We see that the probabilities all form normal distributions with mean values shifted as expected according to the modifications of the rates (higher arrival rates lead to a larger number of clients in the system, higher departure rates lead to a lower number, higher retrial rates again lead to a higher number).

Likewise, Figure 4 investigates the effects of failure rates γ_0 and γ_1 and repair rate γ_2. The effects of the two failure rates are clearly different: increasing the idle failure rate γ_0 has very little effect, while increasing the busy failure rate γ_1 reduces the probability on the left end and increases the probability of the mean value; also increasing the repair rate γ_2 has a similar effect. These results do not really agree well with those of [4] where for $N = 100$ shifts in the distributions towards different mean values are reported. The difference may be due to our small value $N = 6$, the choices of the other parameters, to subtle differences between our models, and, of course, ultimately to the fact that our model considers impatience while theirs does not. Here more investigations are required.

Figure 6 now investigates the core effect of this paper, the impatience of clients. To exhibit the effect of the Erlang distribution, we give figures for $K = 1$ (exponential distribution, upper diagram) as well as for $K = 5$ (middle diagram) and $K = 9$ (lower diagram). The comparison of the diagrams shows that indeed the exponential distribution shows a behavior that is significantly different from the Erlang distributions with $K \geq 5$; however, while $K = 9$ also differs from $K = 5$, the differences are very minor. From these figures we clearly see that, the shorter the patience span T of a client is, the higher the probability is that the client leaves the system (because the request is not served). While for patience spans $T \geq 50$, the differences become smaller, they are still clearly visible.

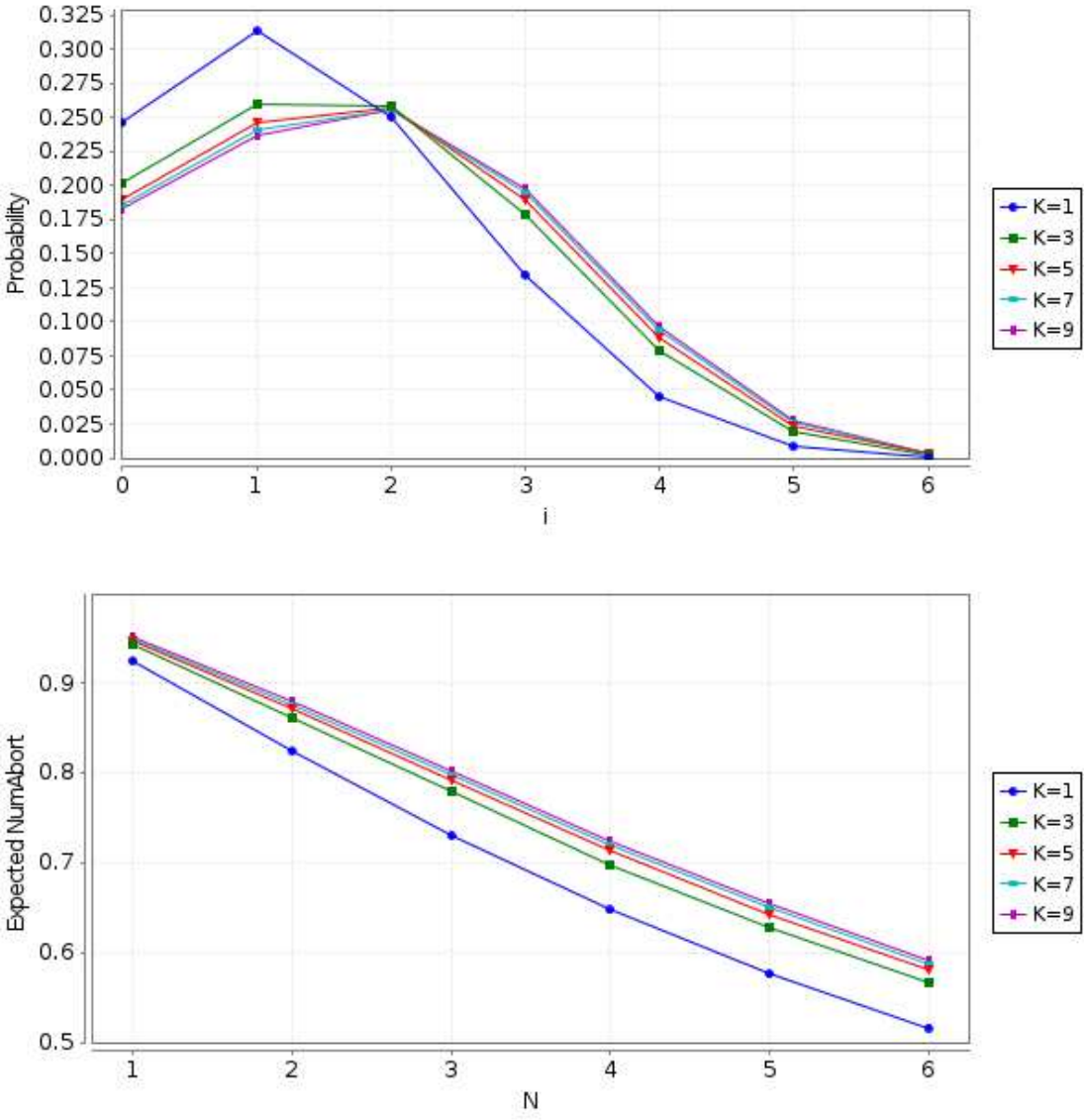


Figure 3: Effects of Erlang shape K

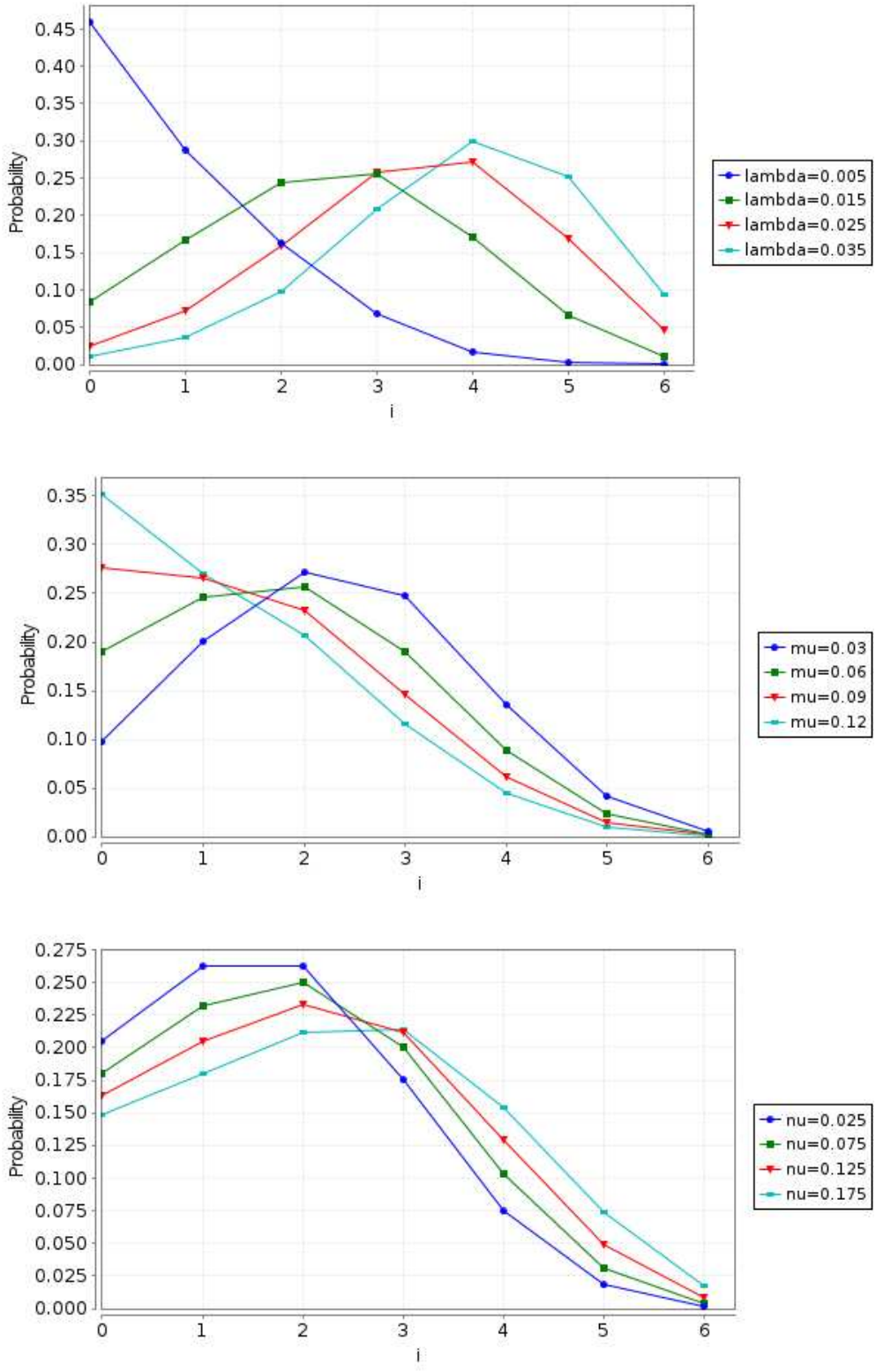


Figure 4: Effect of system rates λ, μ, ν

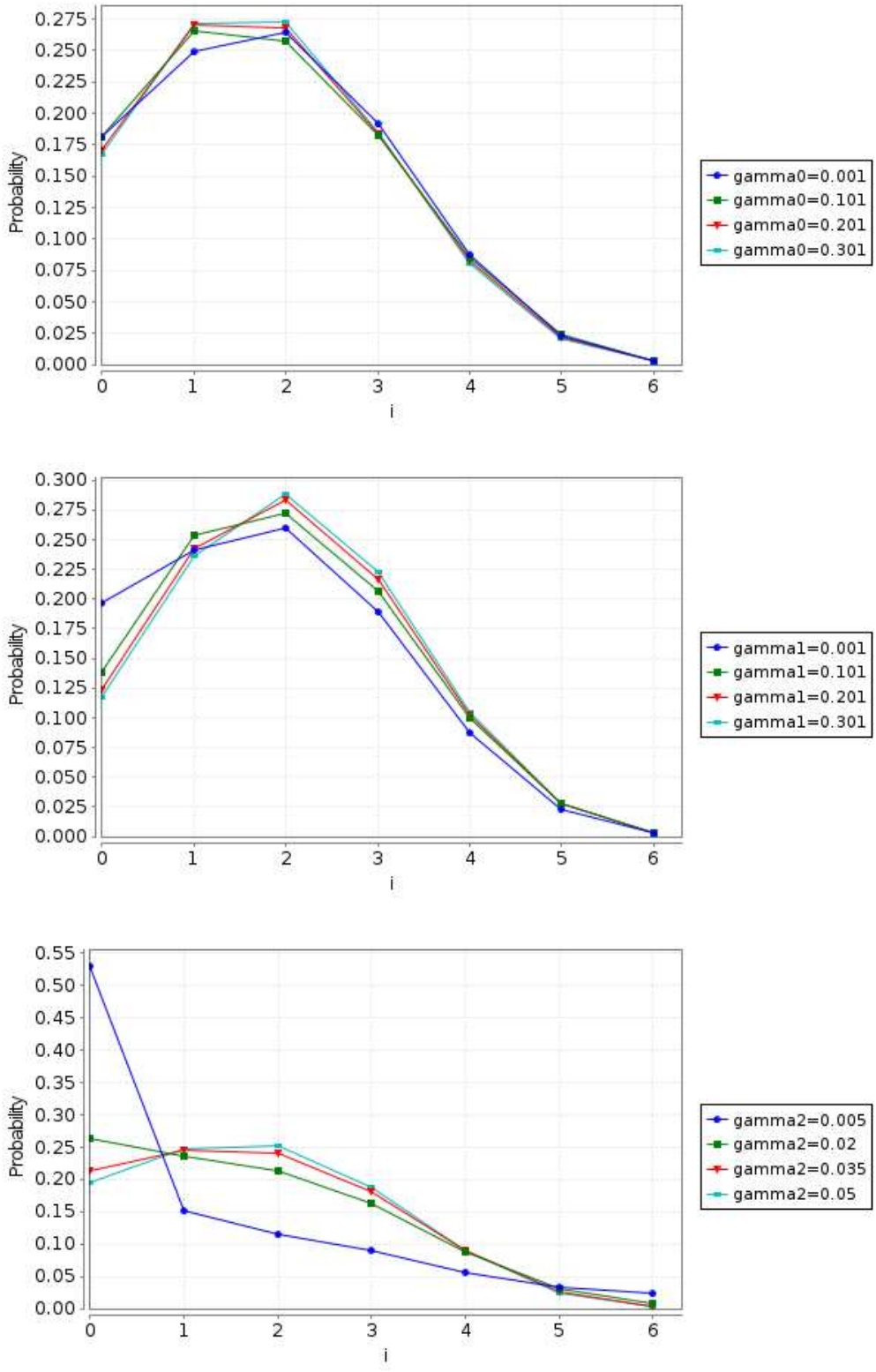


Figure 5: Effect of failure/repair rates γ_0, γ_1, γ_2

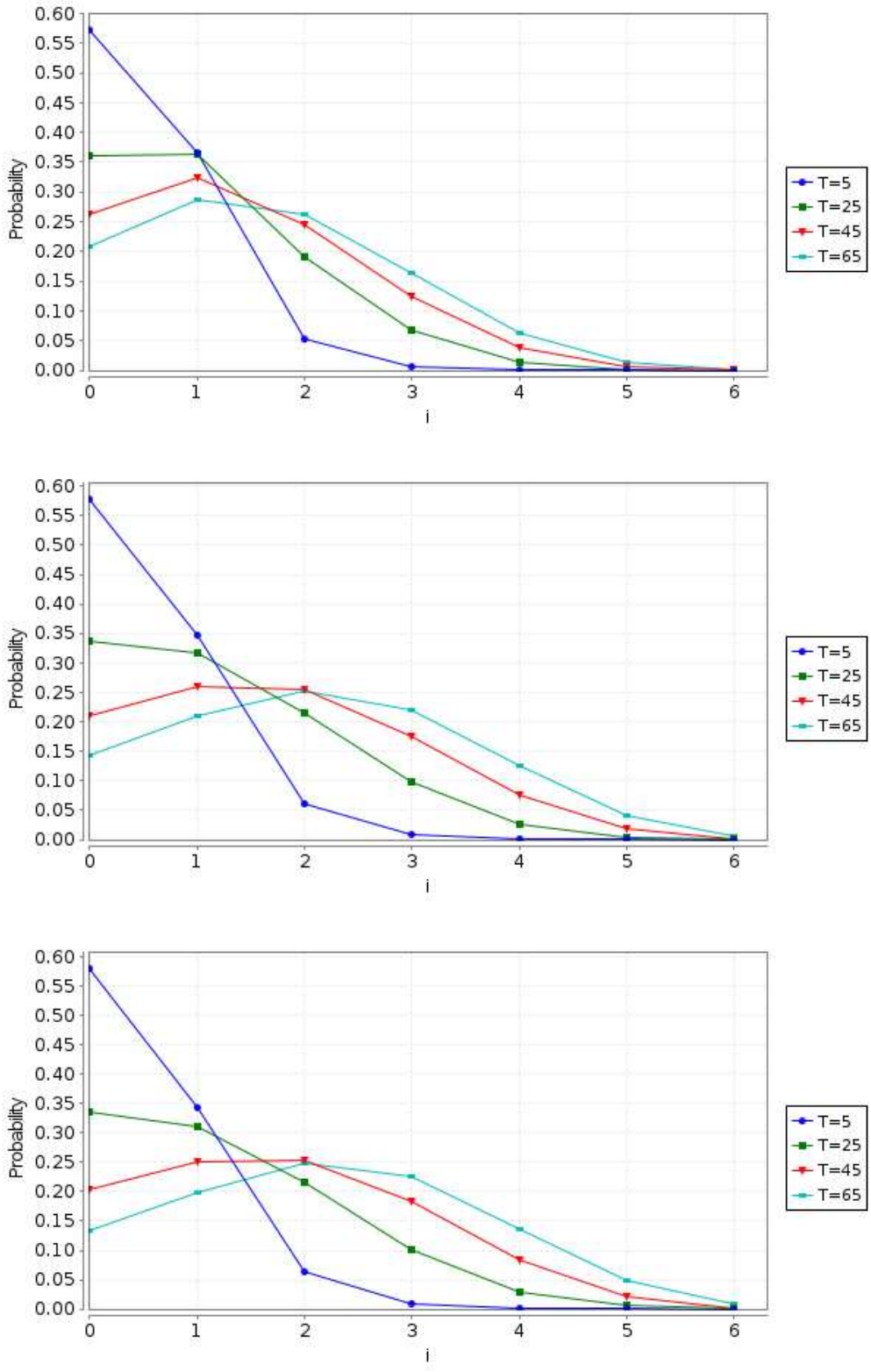


Figure 6: Effect of patience span T for $K = 1$, $K = 5$, and $K = 9$

4 Conclusions

We have investigated how the probabilistic model checker PRISM can be used to model retrieval queueing systems with impatient clients whose patience span is a fixed constant. While principally possible, this approach requires to model the state of every client with an independent "patience timer", an automaton whose transition time follows an Erlang distribution; the approach does therefore not scale but is limited to systems of small size. Nevertheless this work may serve as a starting point for corresponding analytical investigations.

A generally open point is the validation of the presented PRISM model, which may have subtle differences or errors with respect to our informal intentions; here a comparison with results derived from alternative models (e.g., by applying numerical simulation) would be helpful.

References

[1] *fdPRISM*. 2018. URL: https://www.fi.muni.cz/~xrehak/fdPRISM/.

[2] Ľuboš Korenčiak. "Parameter Synthesis in Continuous-Time Stochastic Systems". PhD thesis. Masaryk University, Faculty of Informatics, Brno, 2018. URL: https://is.muni.cz/th/zaes9.

[3] Ľuboš Korenčiak, Vojtěch Řehák, and Adrian Farmadin. "Extension of PRISM by Synthesis of Optimal Timeouts in Fixed-Delay CTMC". In: *Integrated Formal Methods*. Ed. by Erika Ábrahám and Marieke Huisman. Vol. 9681. Lecture Notes in Computer Science. Springer, 2016, pp. 130–138. ISBN: 978-3-319-33693-0. DOI: 10.1007/978-3-319-33693-0_9.

[4] A. Kuki, T.Berczes, J.Sztrik, and A. Kvach. "Numerical Analysis of Retrial Queueing Systems with Conflict of Customers and an Unreliable Server". In: *Journal of Mathematical Sciences* 237.5 (Mar. 2019), pp. 673–683. DOI: 10.1007/s10958-019-04193-1.

[5] 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. DOI: 10.1007/978-3-642-22110-1_47.

[6] Hamza Nemouchi and János Sztrik. "Performance Evaluation of Finite-Source Cognitive Radio Networks with Collision Using Simulation". In: *8th IEEE International Conference on Cognitive Infocommunications (CogInfoCom 2017)*. Debrecen, Hungary, September 11–14, 2017, pp. 127–131. DOI: 10.1109/CogInfoCom.2017.8268228.

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

[8] *PRISM GSMP*. 2019. URL: https://github.com/muhrik/prism-gsmp.

[9] Wolfgang Schreiner, Tamas Berczes, and Janos Sztrik. “Probabilistic Model Checking on HPC Systems for the Performance Analysis of Mobile Networks”. In: *Annales Mathematicae et Informaticae* 43 (2014), pp. 123–144. URL: http://ami.ektf.hu/uploads/papers/finalpdf/AMI_43_from123to144.pdf.

[10] Patrick Wüchner, János Sztrik, and Hermann de Meer. “Finite-source M/M/S retrial queue with search for balking and impatient customers from the orbit”. In: *Computer Networks* 53.8 (2009). Performance Modeling of Computer Networks: Special Issue in Memory of Dr. Gunter Bolch, pp. 1264–1273. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2009.02.015.

A The PRISM Specification

A.1 The System Model

```
// ------------------------------------------------------------------
// ConstantTime.prism
// constant time delay (CTMC version with Erlang approximation)
//
// a finite source system with a single non-reliable server and
// customer impatience with constant time
// (here approximated by Erlang distribution)
//
// (c) 2019, Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
// Research Institute for Symbolic Computation, Johannes Kepler
// University, Linz, Austria (https://www.risc.jku.at)
// ------------------------------------------------------------------

// continuous time markov chain (ctmc) model
ctmc

// ------------------------------------------------------------------
// parameters
// ------------------------------------------------------------------

// number of clients (maximum 10, otherwise model has to be adapted)
const int N;

const double lambda; // arrival rate
const double mu;     // service rate
const double nu;     // retrial rate

const double gamma0; // server failure rate in idle state
const double gamma1; // server failure rate in busy state
const double gamma2; // repair rate

const double T; // duration of patience
const int K;    // shape of Erlang distribution (=1: exponential)

const double infinity = 9999; // infinity rate

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

module Server
  s: [0..3] init 0; // 0: idle, 1: busy, 2: collision, 3: failed

  // evict server due to collision
  [evict] s = 2 -> infinity : (s' = 0);

  // evict server due to failure and then repair it
  [evict]  s = 0 -> gamma0 : (s' = 3);
  [evict]  s = 1 -> gamma1 : (s' = 3);
  [repair] s = 3 -> gamma2 : (s' = 0);
```

```
// interaction with client 1
[source1] N >= 1 & s = 0 ->      (s' = 1);
[orbit1]  N >= 1 & s = 0 ->      (s' = 1);
[exit1]   N >= 1 & s = 1 -> mu : (s' = 0);
[scoll1]  N >= 1 & s = 1 ->      (s' = 2);
[ocoll1]  N >= 1 & s = 1 ->      (s' = 2);
[scoll1]  N >= 1 & s = 3 ->      true;

// interaction with client 2
[source2] N >= 2 & s = 0 ->      (s' = 1);
[orbit2]  N >= 2 & s = 0 ->      (s' = 1);
[exit2]   N >= 2 & s = 1 -> mu : (s' = 0);
[scoll2]  N >= 2 & s = 1 ->      (s' = 2);
[ocoll2]  N >= 2 & s = 1 ->      (s' = 2);
[scoll2]  N >= 2 & s = 3 ->      true;

// interaction with client 3
[source3] N >= 3 & s = 0 ->      (s' = 1);
[orbit3]  N >= 3 & s = 0 ->      (s' = 1);
[exit3]   N >= 3 & s = 1 -> mu : (s' = 0);
[scoll3]  N >= 3 & s = 1 ->      (s' = 2);
[ocoll3]  N >= 3 & s = 1 ->      (s' = 2);
[scoll3]  N >= 3 & s = 3 ->      true;

// interaction with client 4
[source4] N >= 4 & s = 0 ->      (s' = 1);
[orbit4]  N >= 4 & s = 0 ->      (s' = 1);
[exit4]   N >= 4 & s = 1 -> mu : (s' = 0);
[scoll4]  N >= 4 & s = 1 ->      (s' = 2);
[ocoll4]  N >= 4 & s = 1 ->      (s' = 2);
[scoll4]  N >= 4 & s = 3 ->      true;

// interaction with client 5
[source5] N >= 5 & s = 0 ->      (s' = 1);
[orbit5]  N >= 5 & s = 0 ->      (s' = 1);
[exit5]   N >= 5 & s = 1 -> mu : (s' = 0);
[scoll5]  N >= 5 & s = 1 ->      (s' = 2);
[ocoll5]  N >= 5 & s = 1 ->      (s' = 2);
[scoll5]  N >= 5 & s = 3 ->      true;

// interaction with client 6
[source6] N >= 6 & s = 0 ->      (s' = 1);
[orbit6]  N >= 6 & s = 0 ->      (s' = 1);
[exit6]   N >= 6 & s = 1 -> mu : (s' = 0);
[scoll6]  N >= 6 & s = 1 ->      (s' = 2);
[ocoll6]  N >= 6 & s = 1 ->      (s' = 2);
[scoll6]  N >= 6 & s = 3 ->      true;

// interaction with client 7
[source7] N >= 7 & s = 0 ->      (s' = 1);
[orbit7]  N >= 7 & s = 0 ->      (s' = 1);
[exit7]   N >= 7 & s = 1 -> mu : (s' = 0);
[scoll7]  N >= 7 & s = 1 ->      (s' = 2);
```

```
  [ocoll7]  N >= 7 & s = 1 ->      (s' = 2);
  [scoll7]  N >= 7 & s = 3 ->      true;

  // interaction with client 8
  [source8] N >= 8 & s = 0 ->      (s' = 1);
  [orbit8]  N >= 8 & s = 0 ->      (s' = 1);
  [exit8]   N >= 8 & s = 1 -> mu : (s' = 0);
  [scoll8]  N >= 8 & s = 1 ->      (s' = 2);
  [ocoll8]  N >= 8 & s = 1 ->      (s' = 2);
  [scoll8]  N >= 8 & s = 3 ->      true;

  // interaction with client 9
  [source9] N >= 9 & s = 0 ->      (s' = 1);
  [orbit9]  N >= 9 & s = 0 ->      (s' = 1);
  [exit9]   N >= 9 & s = 1 -> mu : (s' = 0);
  [scoll9]  N >= 9 & s = 1 ->      (s' = 2);
  [ocoll9]  N >= 9 & s = 1 ->      (s' = 2);
  [scoll9]  N >= 9 & s = 3 ->      true;

  // interaction with client 10
  [source10] N >= 10 & s = 0 ->      (s' = 1);
  [orbit10]  N >= 10 & s = 0 ->      (s' = 1);
  [exit10]   N >= 10 & s = 1 -> mu : (s' = 0);
  [scoll10]  N >= 10 & s = 1 ->      (s' = 2);
  [ocoll10]  N >= 10 & s = 1 ->      (s' = 2);
  [scoll10]  N >= 10 & s = 3 ->      true;
endmodule

module Client1
  c1: [0..2] init 0; // 0: source, 1: server, 2: orbit
  t1: [0..K] init 0; // tick counter when in orbit

  // interaction with server
  [source1] c1 = 0 -> lambda : (c1' = 1);
  [scoll1]  c1 = 0 -> lambda : (t1' = 1);
  [orbit1]  c1 = 2 -> nu     : (c1' = 1) & (t1' = 0);
  [ocoll1]  c1 = 2 -> nu     : true;
  [exit1]   c1 = 1 ->          (c1' = 0);

  // go to orbit due to collision or failure
  [evict] c1 = 0 & t1 > 0  -> (c1' = 2);
  [evict] c1 = 0 & t1 = 0  -> true;
  [evict] c1 = 1           -> (c1' = 2) & (t1' = 1);
  [evict] c1 = 2           -> true;

  // in orbit countdown to abortion
  [tick1]  c1 = 2 & t1 < K -> K/T : (t1' = t1+1);
  [abort1] c1 = 2 & t1 = K -> K/T : (c1' = 0) & (t1' = 0);
endmodule

module Client2 = Client1
[ c1=c2, t1=t2, source1=source2, scoll1=scoll2, ocoll1=ocoll2,
  orbit1=orbit2, exit1=exit2, tick1=tick2, abort1=abort2 ]
endmodule
```

```
module Client3 = Client1
[ c1=c3, t1=t3, source1=source3, scoll1=scoll3, ocoll1=ocoll3,
  orbit1=orbit3, exit1=exit3, tick1=tick3, abort1=abort3 ]
endmodule

module Client4 = Client1
[ c1=c4, t1=t4, source1=source4, scoll1=scoll4, ocoll1=ocoll4,
  orbit1=orbit4, exit1=exit4, tick1=tick4, abort1=abort4 ]
endmodule

module Client5 = Client1
[ c1=c5, t1=t5, source1=source5, scoll1=scoll5, ocoll1=ocoll5,
  orbit1=orbit5, exit1=exit5, tick1=tick5, abort1=abort5 ]
endmodule

module Client6 = Client1
[ c1=c6, t1=t6, source1=source6, scoll1=scoll6, ocoll1=ocoll6,
  orbit1=orbit6, exit1=exit6, tick1=tick6, abort1=abort6 ]
endmodule

module Client7 = Client1
[ c1=c7, t1=t7, source1=source7, scoll1=scoll7, ocoll1=ocoll7,
  orbit1=orbit7, exit1=exit7, tick1=tick7, abort1=abort7 ]
endmodule

module Client8 = Client1
[ c1=c8, t1=t8, source1=source8, scoll1=scoll8, ocoll1=ocoll8,
  orbit1=orbit8, exit1=exit8, tick1=tick8, abort1=abort8 ]
endmodule

module Client9 = Client1
[ c1=c9, t1=t9, source1=source9, scoll1=scoll9, ocoll1=ocoll9,
  orbit1=orbit9, exit1=exit9, tick1=tick9, abort1=abort9 ]
endmodule

module Client10 = Client1
[ c1=c10, t1=t10, source1=source10, scoll1=scoll10, ocoll1=ocoll10,
  orbit1=orbit10, exit1=exit10, tick1=tick10, abort1=abort10 ]
endmodule

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

rewards "NumClient"
  c1  > 0 : 1;
  c2  > 0 : 1;
  c3  > 0 : 1;
  c4  > 0 : 1;
  c5  > 0 : 1;
  c6  > 0 : 1;
  c7  > 0 : 1;
  c8  > 0 : 1;
```

```
  c9  > 0 : 1;
  c10 > 0 : 1;
endrewards

rewards "NumExit"
  [exit1]  true : 1;
  [exit2]  true : 1;
  [exit3]  true : 1;
  [exit4]  true : 1;
  [exit5]  true : 1;
  [exit6]  true : 1;
  [exit7]  true : 1;
  [exit8]  true : 1;
  [exit9]  true : 1;
  [exit10] true : 1;
endrewards

rewards "NumAbort"
  [abort1]  true : 1;
  [abort2]  true : 1;
  [abort3]  true : 1;
  [abort4]  true : 1;
  [abort5]  true : 1;
  [abort6]  true : 1;
  [abort7]  true : 1;
  [abort8]  true : 1;
  [abort9]  true : 1;
  [abort10] true : 1;
endrewards

// ---------------------------------------------------------------------
// end of model
// ---------------------------------------------------------------------
```

A.2 The CSL Queries

```
// number of customers in system
"NC": R{"NumClient"}=? [ S ] ;

// number of successful exits from server
"NE": R{"NumExit"}=? [ S ] ;

// number of aborts from orbit
"NA": R{"NumAbort"}=? [ S ] ;

// probability for successful exit
"PE": 1/(1+("NA"/"NE"));

// average sojourn time
"T":  "NC"/(lambda*(N-"NC")*"PE");

// probability that i clients are in system
const int i;
```

```
"Pi": S=? [ i = min(c1,1)+min(c2,1)+min(c3,1)+min(c4,1)+min(c5,1)
              +min(c6,1)+min(c7,1)+min(c8,1)+min(c9,1)+min(c10,1) ];
```