Experiments with Measuring Time in PRISM 4.0*

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

March 1, 2013

Abstract

We report on experiments with the probabilistic symbol model checker PRISM 4.0 on evaluating the response times of client/server system models. In earlier work, we described such systems by queue models where only the number of pending requests was stored and measured. Under certain assumptions, we could then apply results from queueing theory (Little's Law) to derive the time that a request spends until getting processed. In this report, we first revisit this approach and substantiate it by a more detailed analytic analysis. We then investigate various possibilities how times can be directly measured in PRISM in continuous and discrete time models and in this process also investigate various features that have been introduced in the new version 4.0 of PRISM that were not yet available for our earlier work. Finally, we apply our insights to measuring time explicitly in client/server models and compare the results to those that we have previously derived by indirect means.

*Supported by the project 85öu8 of the Stiftung Aktion Österreich-Ungarn.

Contents

1. Introduction

In this report we investigate the use of the probabilistic symbolic model checker PRISM 4.0 [6, 8] for measuring response times in client/server system models. We build upon earlier work of ours [3, 1] where we used the earlier version PRISM 3.x for describing and analyzing such systems by queue models; however response times could be derived from these models only in a rather indirect way by applying Little's Law which is well known in queueing theory [4]. However, while we have provided semi-formal arguments whey the law could be also applied to our PRISM models, there has remained an uncomfortable gap between the actual measurements and the derived conclusions.

Our goal is therefore to substantiate the previous result by a more detailed analysis but also to investigate more direct ways of measuring response times provided by PRISM considering also the features newly introduced in version 4.0. In this process, we do not restrict ourselves to queueing models but seek to apply also more direct models as have become available by probabilistic/stochastic model checking [7]. In fact, while previously the performance modeling analysis of computing systems has been the realm of classical queuing theory, formal methods have provided new possibilities[9] that we would like to exploit, in particular the ability to describe systems as finite state machines in much larger detail than classical queueing networks [4].

The remainder of this report is organized as follows: in Section 2, we perform a more detailed analysis of the queue models of client/server systems used in our previous work. Section 3 investigates various alternatives for dealing with time in PRISM, considering both continuous and discrete time models. In Section 4, we apply our insight to modeling and analyzing client/server systems in such a way that response times can be directly by PRISM in a direct way. In Section 5, we present the conclusions of our work.

We assume that the reader is familiar with the basic notions of PRISM, for background information see [8, 6, 7].

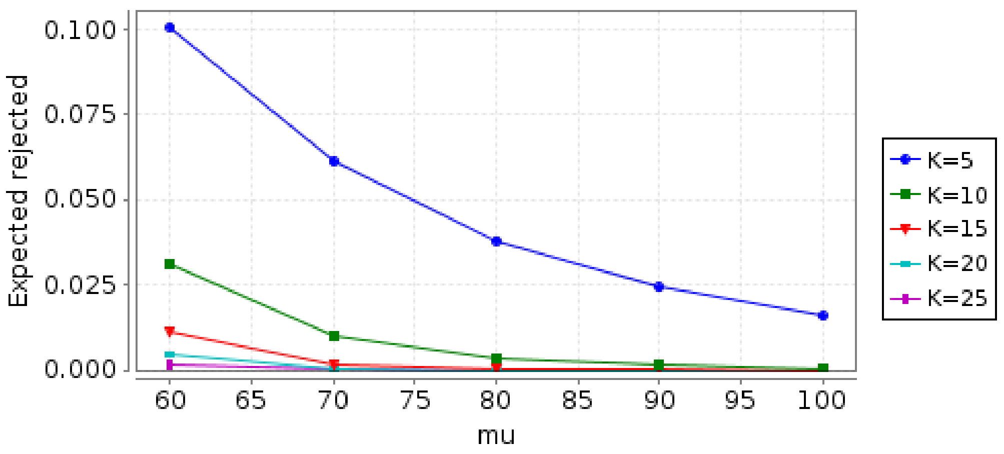
2. A Queue-Based Model of a Client/Server System

We start with the analysis of a simple model of a client that generates requests with rate λ which are processed by a server with rate μ; client and server are connected by a bounded queue of capacity K. The corresponding PRISM "Continuous Time Markov Chain" (CTMC) model is depicted in Appendix A.

To determine the mean time T that a request has to wait until being serviced, we pursue in this section the strategy we have devised in [3]: we measure the average number of requests N in the queue and apply "Little's Theorem" to compute $T = N/\lambda$. However, in this form the theorem actually applies only to a system with unbounded queues. To apply it to the case of bounded queues, we must first determine the probability p that a generated request is rejected by the queue (because it is full) and then compute $T = N/(\lambda \cdot (1-p))$.

Rejection Probability To determine the rejection probability p, we introduce in the PRISM model a reward structure

```
    rewards "rejected"
```



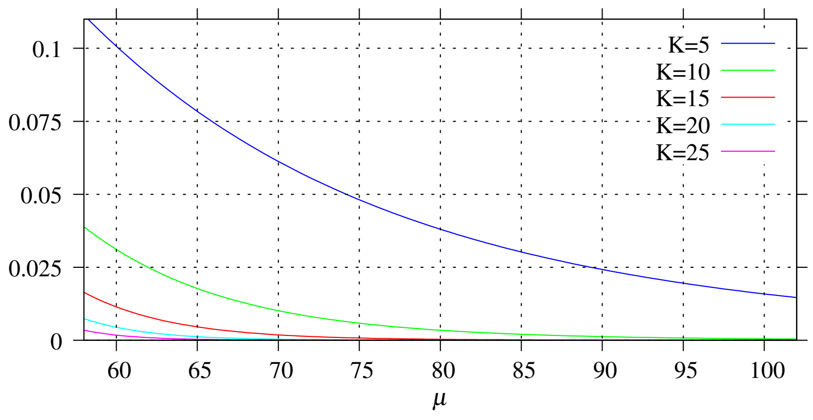


Figure 1: Analyzing the Rejection Probability

```
    !accepted : 1;
  endrewards
```

that assigns to every state of the system where the queue has not accepted a request reward 1 and to every other state reward 0. Then the CSL query

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

computes the rejection probability p as the corresponding steady-state reward, i.e., as the average value of the reward in the long run. The top diagram in Figure 1 (generated with the help of the "Experiments" feature of PRISM) illustrates the course of the measured probability $p(\lambda,\mu,K)$ for fixed arrival rate $\lambda = 50$, various acceptance rates $\mu \in [60,100]$, and queue bounds $K \in \{5,10,15,20,25\}$; since $\lambda < \mu$, the system reaches for all of these parameter values a stable state with $p < 1$. We see that for $K > 5$ we have $p < 0.1$ and for $K \geq 15$ we even get $p < 0.01$.

Since we can determine from queueing theory analytically

$$p = \frac{1-\rho}{1-\rho^{K+1}}\rho^K$$

with utilization $\rho = \lambda/\mu$, it is also possible to validate the results derived from the PRISM model. For this purpose, we plot the curve for the analytically derived function p as shown in the bottom of Figure 1. We see that the measured values fit to the analytically derived ones with high accuracy.

Number of Waiting Requests Likewise, to determine the number N of waiting requests, we introduce the reward structure

```
rewards "waiting"
  true : waiting;
endrewards
```

that assigns to every state of the system as a reward the number of elements in the queue. The CSL query

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

then computes N as the corresponding steady-state reward. The results of the query (as before, for $\lambda = 50$ and various values of μ and K) are depicted in the top diagram of Figure 2.

Again we can validate the measured results by an analytical analysis based on the results of queueing theory. For an unbounded queue, we have $N = \rho/(1-\rho) = \lambda/(\mu-\lambda)$. For a bounded queue with size K we have

$$N = \frac{\rho}{1-\rho} - \frac{(K+1)\cdot\rho^{K+1}}{1-\rho^{K+1}}$$

By plotting the corresponding curve $N(\lambda,\mu,K)$ in the bottom of Figure 2, we again see that the measured values fit to the analytically derived ones very well. By comparing with the value for the unbounded queue model ($K=\infty$), we also learn that for $K \geq 25$ there is no substantial difference of the results for that of an unbounded queue any more, as already suggested by the convergence of the curves for $K = 20$ and $K = 25$.

The Waiting Time Finally, with the help of the CSL queries

```
"time0": "waiting"/lambda;
"time1": "waiting"/(lambda*(1-"rejected"));
```

it becomes possible to compute the waiting time T, once as the unbounded queue approximation $T = 1/N$ and once as the exact bounded queue value $T = N/((1-p)\cdot\lambda)$ (this kind of queries that combines multiple reward computations to which we can refer by names has become possible with PRISM 4.0; in the older PRISM versions we used in [3] the computation had to be performed by external means). The course of T for these two computations is depicted in the two upper diagrams of Figure 3. We see that for $K \geq 15$ both diagrams give almost identical results (since here $p < 0.01$).

Once again we validate the measured results by an analytical analysis. For an unbounded queue, we have $T = N/\lambda = 1/(\mu-\lambda)$. For a bounded queue with size K we have $T = N/(\lambda\cdot(1-p))$. By plotting the corresponding curve $T(\lambda,\mu,K)$ (as before, for $\lambda = 50$ and various

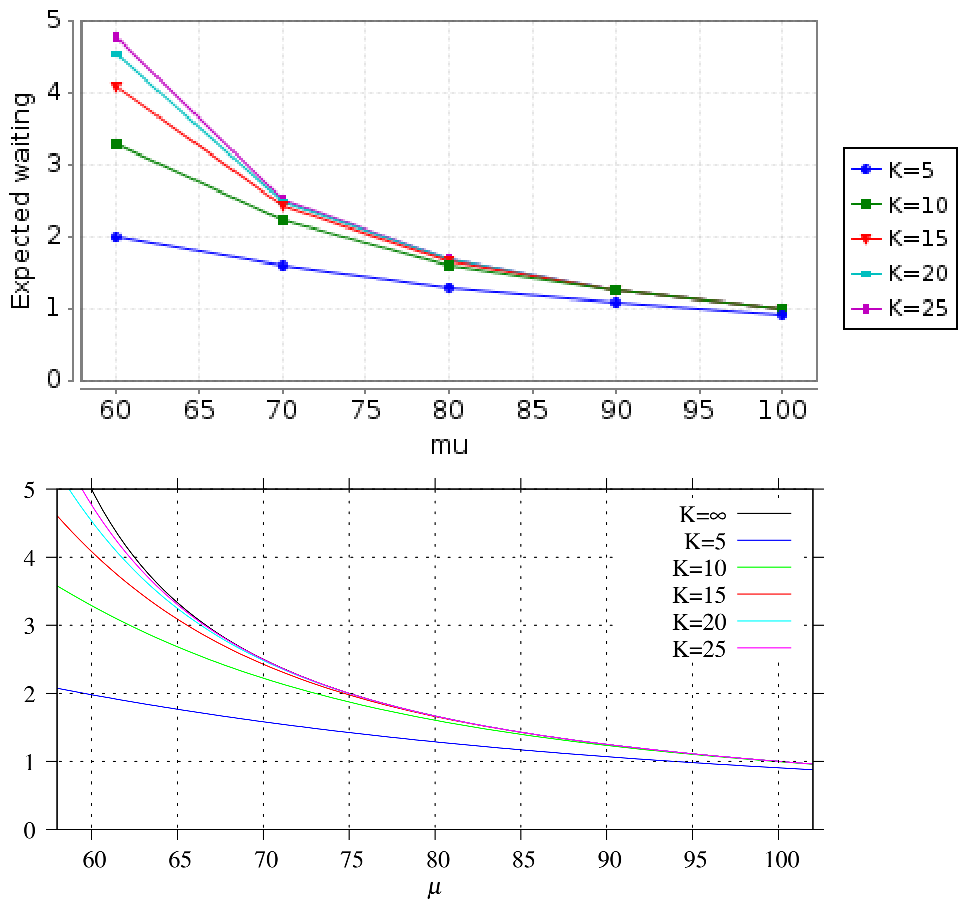


Figure 2: Analyzing the Number of Waiting Requests

values of μ and K) in the bottom of Figure 3, we again see that the measured values fit to the analytically derived ones very well. By comparing with the value for the unbounded queue model ($K = \infty$), we also learn that for $K \geq 25$ there is no substantial difference to the waiting time of an unbounded queue any more, as already suggested by the convergence of the curves for $K = 20$ and $K = 25$.

Summarizing, we see that PRISM can be used to model and analyze systems with bounded queues with high accuracy. For this purpose, we have complemented our earlier work [3] by validating the measured results against the analytic results derived from queueing theory; we have thus also substantiated our earlier claim that the convergence of the measurement curves for growing queue sizes in a model with bounded queues illustrates the limit case of a system with unbounded queues.

From the practical point of view, we have learned that, by the new feature of nested reward computation introduced in PRISM 4.0, the interesting measures can be determined directly within PRISM without resorting to external tools any more.

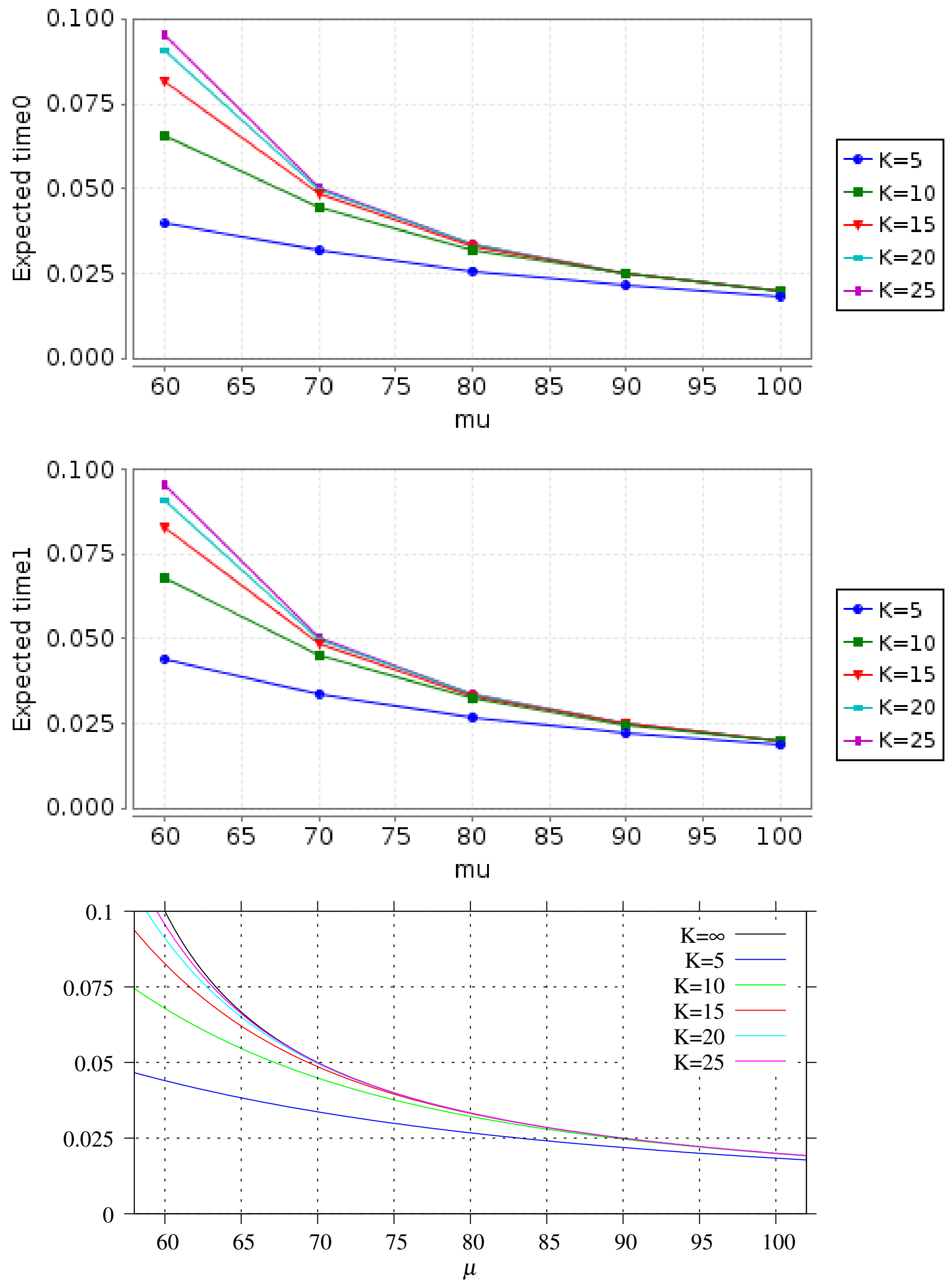


Figure 3: Analyzing the Waiting Time

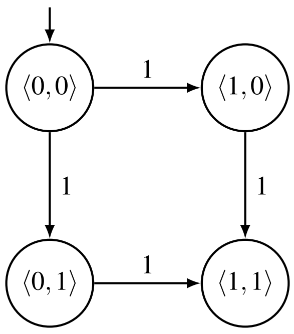
3. Approaches to Explicit Time Measurement

In the Section 2, we have measured the time that a request has to wait to be serviced only indirectly by describing the client/server system as a queueing model, measuring the number of elements in the queue, and applying Little's Theorem to determine the waiting time, corresponding to our earlier approaches [3]. In this section, we will extend our previous work by investigating more direct means to measure the waiting time of requests using the different models supported by PRISM. Our running example is a simple model of form

```
module M1
  state1: [0..1] init 0;
  [] state1 = 0 -> (state1' = 1);
endmodule

module M2
  state2: [0..1] init 0;
  [] state2 = 0 -> (state2' = 1);
endmodule
```

with two components independently traversing from state 0 to state 1 (and staying there); we are interested to measure how long it takes component 1 to make the transition.

The corresponding system has four states



where each state is labeled with the tuple $\langle state_0, state_1 \rangle$ of the values of the state variables and every transition has rate 1 (unless stated explicitly otherwise, we assume CTMC models).

3.1. A Digital Clock with State

The simplest approach is to equip the system with a digital clock that ticks with rate M and to count the number of ticks encountered so far. to make the number of states finite, however, we have to bound the number of ticks c by some value N (see Appendix B.1 for the full code of the model):

```
module Clock
  c: [0..N] init 0;
  [] true -> M : (c' = min(c+1,N));
endmodule
```

$$\begin{tikzcd}
\langle 0,0\rangle \arrow[r, "1"] \arrow[d, "1"] & \langle 1,0\rangle \arrow[d, "1"] \\
\langle 0,1\rangle \arrow[r, "1"] & \langle 1,1\rangle
\end{tikzcd}$$

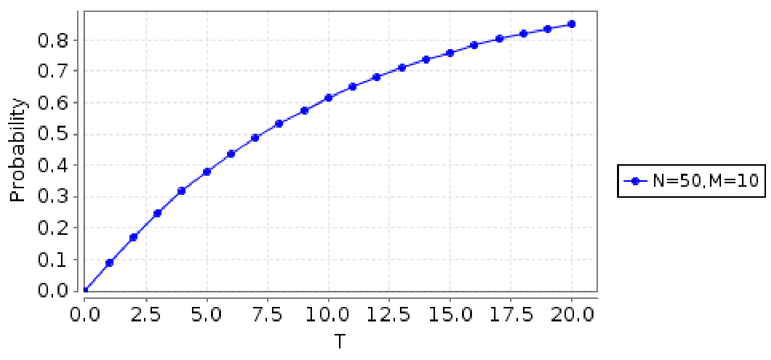


Figure 4: A Digital Clock

By a CSL query

```
P=? [ c < T U state1 = 1 ];
```

we can determine the probability that c remains below T until the state transition occurs. Figure 4 depicts the course of this probability for growing T.

While conceptually simple, this approach has also several disadvantages:

1. It increases the state space of the system by a factor N, which may make model checking prohibitive.
2. It does actually not compute a particular time but only a probability that an event occurs by a certain instance of time.
3. The number of ticks is bounded; this approach does not work on the long run for a non-terminating system where the average time between recurring pairs of events is of interest.

In any case, PRISM provides better approaches.

3.2. A Digital Clock without State

Rather than equipping the digital clock with explicit state, we may also let it just “tick” (see Appendix B.2 for the full model):

```
module Clock
  [tick] true -> M : true;
endmodule
```

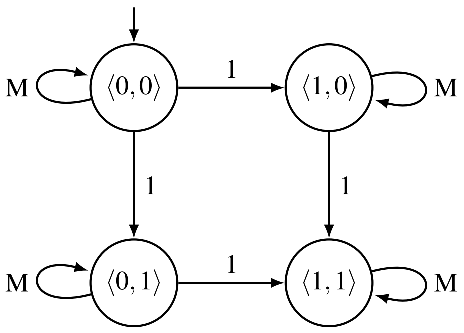
We can associate to every tick a “reward” that corresponds to the average time of a tick

```
rewards
  [tick] true : 1/M;
endrewards
```

A CSL query may thus compute the average reward accumulated on an execution path until the state transition occurs:

```
    R=? [ F state1 = 1 ];
```

PRISM gives for this query a result of approximately 1 (accurate up to 15 digits) which can be explained as follows: the analyzed system has structure



where each state has a transition to itself with rate M corresponding to the tick of the clock.

Since we have one transition from state $\langle 0,0\rangle$ to itself with rate M and two transitions to other states with rate 1 each, a transition leaves state $\langle 0,0\rangle$ with probability $p = 2/(M+2)$. According to the laws of a geometric distribution, the expected number of iterations before leaving this state is $(1-p)/p = M/2$; since each iteration accumulates reward $1/M$, the expected reward accumulated before leaving this state is $(M/2)\cdot(1/M) = 1/2$. Likewise the probability of leaving state $\langle 0,1\rangle$ is $p = 1/(M+1)$, the expected number of iterations in that state is $(1-p)/p = M$ and the accumulated reward is $M\cdot(1/M) = 1$.

Since both transitions out of state $\langle 0,0\rangle$ have equal rates, both paths $\langle 0,0\rangle \to \langle 1,0\rangle$ and $\langle 0,0\rangle \to \langle 0,1\rangle \to \langle 1,1\rangle$ leading to a desired state have probability $1/2$. The expected reward accumulated up to a desired state is thus

$$\frac{1}{2}\cdot\frac{1}{2}+\frac{1}{2}\cdot(\frac{1}{2}+1) = \frac{1}{4}+\frac{3}{4} = 1$$

This result also meets our intuition, since this is the same result we would have got in a system with only one component and adding a concurrent second component should not delay the execution of the first one.

While this solution does give satisfactory results without the disadvantages of an explicit state representation, it still requires the addition of a concurrent clock component; this is conceptually unsatisfactory, since we actually measure the execution time of the clock rather than the execution time of the components of interest. In the following, we will therefore strive to get rid of the separate clock component.

3.3. No Digital Clock

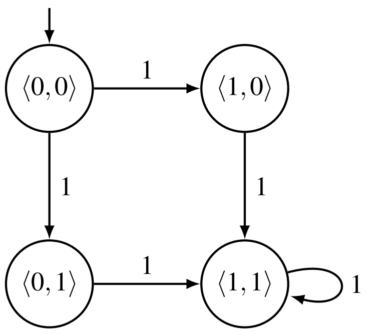
Since we have measured in the previous section the transitions of the clock, our first idea to get rid of the clock is to measure the transitions of the components themselves. We change therefore the model (see the model code in Appendix B.3) to

```
module M1
  state1: [0..1] init 0;
  [step1] state1 = 0 -> (state1' = 1);
endmodule

module M2
  state2: [0..1] init 0;
  [step2] state1 = 1 -> (state2' = 1);
endmodule

rewards "time1"
  [step1] true : 1;
  [step2] true : 1;
endrewards
```

which associates to each transition a reward 1 (the time we can expect to stay in the local module state ignoring the transitions from other modules). Then the query

```
R{"time1"}=? [ F state1 = 1 ];
```

now yields the somewhat surprising result 1.5, different from the value 1.0 measured in the previous section (PRISM also warns that a self loop is added to state $\langle 1,1\rangle$, which does not concern us for our analysis). However, the explanation is not difficult: the analyzed model is



with two paths leading from state $\langle 0,0\rangle$ to the states $\langle 1,0\rangle$ and $\langle 1,1$: the first one accumulates a transition reward 1, the second one accumulates transition reward 2. Since both paths are taken with equal probability the accumulated reward is $0.5\cdot 1+0.5\cdot 2=1.5$.

The problem is that we have accumulated rewards 1 for the transitions leaving state $\langle 0,0\rangle$ while the expectation value for the time spent in this tate is only 0.5 (the reciprocal value of the exit rate of that state). This means we have to refine our reward structure

```
rewards "time2"
  [step1] state2 = 0 : 0.5;
  [step1] state2 = 1 : 1;
  [step2] state1 = 0 : 0.5;
  [step2] state1 = 1 : 1;
endrewards
```

The corresponding query

```
R{"time2"}=? [ F state1 = 1 ];
```

then yields the expected result 1.0 $(= 0.5 \cdot 0.5 + 0.5 \cdot (0.5 + 1))$.

However, the new reward structure requires a global analysis of the model to determine the correct reward for each type of transition according to the state in which the transition is taken; this is awkward and error prone and definitely not what we want.

A better approach provided by PRISM is to associate rewards to *states*: if a value r is associated to a state, this value is actually considered as the *rate* with which the rate is accumulated: if the system spends time t in the state, the total reward accumulated is $r \cdot t$. We therefore define the state reward

```
rewards "time3"
  true : 1;
endrewards
```

The corresponding query

```
R{"time2"}=? [ F state1 = 1 ];
```

then also yields result $1.0(= 0.5 \cdot 0.5 + 0.5 \cdot (0.5 + 1))$.

We therefore conclude that measuring times by rewards associated to *transitions* is only advisable, if these are the transitions of an independently executing “clock”. Without a separate clock component, it is better to measure the rewards associated to the states of the model since here the actual times spent in the states will be accumulated.

3.4. Discrete Time Models

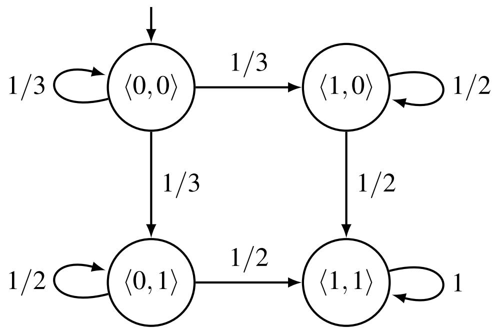
Up to now, we only have considered Continuous Time Markov Chain (CTMC) models; we will now turn our attention towards Discrete Time Markov Chain (DTMC) models. While syntactically similar to CTMC models, the values associated to transitions are here not considered as rates but as *probabilities* that transitions are taken.

Digital Clock A DTMC model with explicit clock (without state) can be constructed as follows (see the model code in Appendix B.4):

```
module M1
  state1: [0..1] init 0;
  [] state1 = 0 -> (state1' = 1);
endmodule

module M2
  state2: [0..1] init 0;
  [] state2 = 0 -> (state2' = 1);
endmodule
```

```
module Clock
  [tick] true -> true;
endmodule
```

In every transition of a module, the total probability of the transition (multiple update possibilities may be given) must be 1; the parallel composition lets each transition be taken with equal probability. The resulting probability distribution is thus as follows:



The probability of leaving state $\langle 0,0\rangle$ is $p = 2/3$, according to the laws of a geometric distribution, the expected number of clock "ticks" in that state is thus $(1-p)/p = 1/2$. Likewise, the probability of leaving state $\langle 0,1\rangle$ is $p = 1/2$, the number of ticks in that state is thus $(1-p)/p = 1$. If, as in the previous CTMC case, we would like state $\langle 0,0\rangle$ time $t = 1/2$ and state $\langle 1,0\rangle$ time $t = 1$, we must use the reward structure

```
rewards
  [tick] state1 = 0 & state2 = 0 : 0.5/0.5;
  [tick] state1 = 0 & state2 = 1 : 1/1;
  [tick] state1 = 1 & state2 = 0 : 1/1;
endrewards
```

i.e. assign reward t/p. With the corresponding query

```
R=? [ F state1 = 1 ];
```

the result 1 (an approximation accurate up to 15 digits is returned).

No Digital Clock As for a discrete model without explicit clock, we can give the following (see Appendix B.5):

```
module M1
  state1: [0..1] init 0;
  [step1] state1 = 0 -> (state1' = 1);
endmodule
```

```
module M2
  state2: [0..1] init 0;
  [step2] state2 = 0 -> (state2′ = 1);
endmodule
```

and assign rewards to it as in the continuous case. With assignments

```
rewards "time1"
  [step1] true : 1;
  [step2] true : 1;
endrewards
```

respectively

```
rewards "time2"
  [step1] state2 = 0 : 0.5;
  [step1] state2 = 1 : 1;
  [step2] state1 = 0 : 0.5;
  [step2] state1 = 1 : 1;
endrewards
```

and the same queries as before, we get again results 1.5 and 1.0. However, this is only because the continuous model had same rates assigned to the transitions outgoing from state $\langle 0,0\rangle$ and thus both branches are taken with equal probability, as in the discrete case shown above. If the outgoing transitions would have different rates, it would not be possible to achieve the same results in the discrete model by assigning rewards to transitions.

Trying to use the same reward structure as in the continuous case

```
rewards "time3"
  true : 1 ;
endrewards
```

the same query as before yields now result 1.5 (rather than 1.0). The reason is that in the discrete case the value 1 assigned to every state is not any more a reward rate but the reward value itself; as a consequence, every passed state contributes 1 to the reward, i.e., the computed reward describes the *number* of states passed ($1 = 0.5 \cdot 1 + 0.5 \cdot 2$). However, if we use the reward structure

```
rewards "time4"
  state1 = 0 & state2 = 0 : 0.5;
  state1 = 1 & state2 = 0 : 0.5;
  state1 = 0 & state2 = 1 : 1;
  state1 = 1 & state1 = 1 : 1;
endrewards
```

that assigns to every state the expected amount of time spent in that state (the reciprocals of the exit rates) we get the values 1.0, as before.

Markov Decision Processes In PRISM, a Markov Decision Process (MDP) is a timed automaton that nondeterministically selects between multiple enabled transitions (without having a certain probability assigned to each selection); the model code for our example is the same as for the DTCM model (except for the keyword `mdp` rather than `dtmc`).

All queries with respect to an MDP have to take all possible executions of the system (all possible resolutions of nondeterminism) separately into account; thus only lower/upper bounds of probabilities and rewards can be computed. For instance, in an MDP with digital clock, we can perform the queries

```
Rmin=? [ F state1 = 1 ];
Rmax=? [ F state1 = 1 ];
```

for the minimum/maximum values of the "clock tick" rewards assigned to the system; the results are 0.0 and ∞, respectively (since no ticks as well as infinitely many ticks might be performed).

In an MDP without digital clock, we can perform analogous queries with respect to the rewards assigned by state or transition rewards:

- for reward structures "time1" and "time3" (counting transitions, respectively states), we have $R \in [1.0, 2.0]$
- for reward structures "time2" and "time4" (accumulating the adjusted transition and state rewards), we have $R \in [0.5, 1.5]$

In both cases, the minimum value corresponds to the path with 1 transition, the maximum value corresponds to the path with 2 transitions.

Probabilistic Timed Automata PRISM 4.0 introduces support for the model of Probabilistic Timed Automata (PTA), essentially MDP models where an additional type "clock" can be used to constrain the behavior of the system:

1. A references to a clock in a transition guard can constrain the time interval when a transition *may* occur.
2. A clock invariant can ensure that a transition *must* occur in a certain time interval.
3. Transition updates may reset clock values to fixed integer values (this is typically used to reset the clock to 0).

We could thus describe our example system as (see Appendix B.6)

```
module M1
  state1: [0..1] init 0;
  c1: clock;
  invariant state1 = 0 => c1 <= T2 endinvariant
  [step1] state1 = 0 & T1 <= c1 -> (state1' = 1);
endmodule
```

```
module M2
  state2: [0..1] init 0;
  c1: clock;
  invariant state2 = 0 => c2 <= T2 endinvariant
  [step2] state2 = 0 & T1 <= c2 -> (state2′ = 1);
endmodule
```

which says that the transition from state 0 to state 1 occurs in time interval $[T_1, T_2]$.

The analysis of PTAs is currently restricted to computing the minimum/maximum probability/reward of (bounded) reachability properties for all possible transition times allowed by the clock constraints (properties and reward structures must not refer to clock values). For instance, the queries

```
Pmin=? [ F<=T state1=1 ]
Rmax=? [ F state1=1 ]
```

ask what is the minimum probability that the transition to state 1 occurs by time T or what is the maximum reward accumulated by the time when the transition to state 1 is performed.

Since clocks are integers, we scale our time unit by a factor of 100 (i.e., 100 clock cycles correspond to 1 time unit) and compute for $T_1 = 80$ and $T_2 = 120$ the minimum/maximum values of some of the rewards also computed for MDPS:

- for "time1", we have $R \in [100, 200]$;
- for "time2", we have $R \in [50, 150]$;
- for "time3", we have $R \in [200, 300]$;

The results for "time1" and "time2" (using transition rewards) correspond to those derived for the MDP model. However, we are unable explain the result for "time3". To check our understanding of state rate accumulation in PTAs, we have used the simple PTA model

```
pta;

const int T1; const int T2;
module M
  state: [0..1] init 0;
  c: clock;
  invariant
    state = 0 => (c <= T2)
  endinvariant
  [e1] state = 0 & T1 <= c -> (state′ = 1);
  [e2] state = 1 -> (state′ = 1);
endmodule

rewards "time"
  true : 1;
endrewards
```

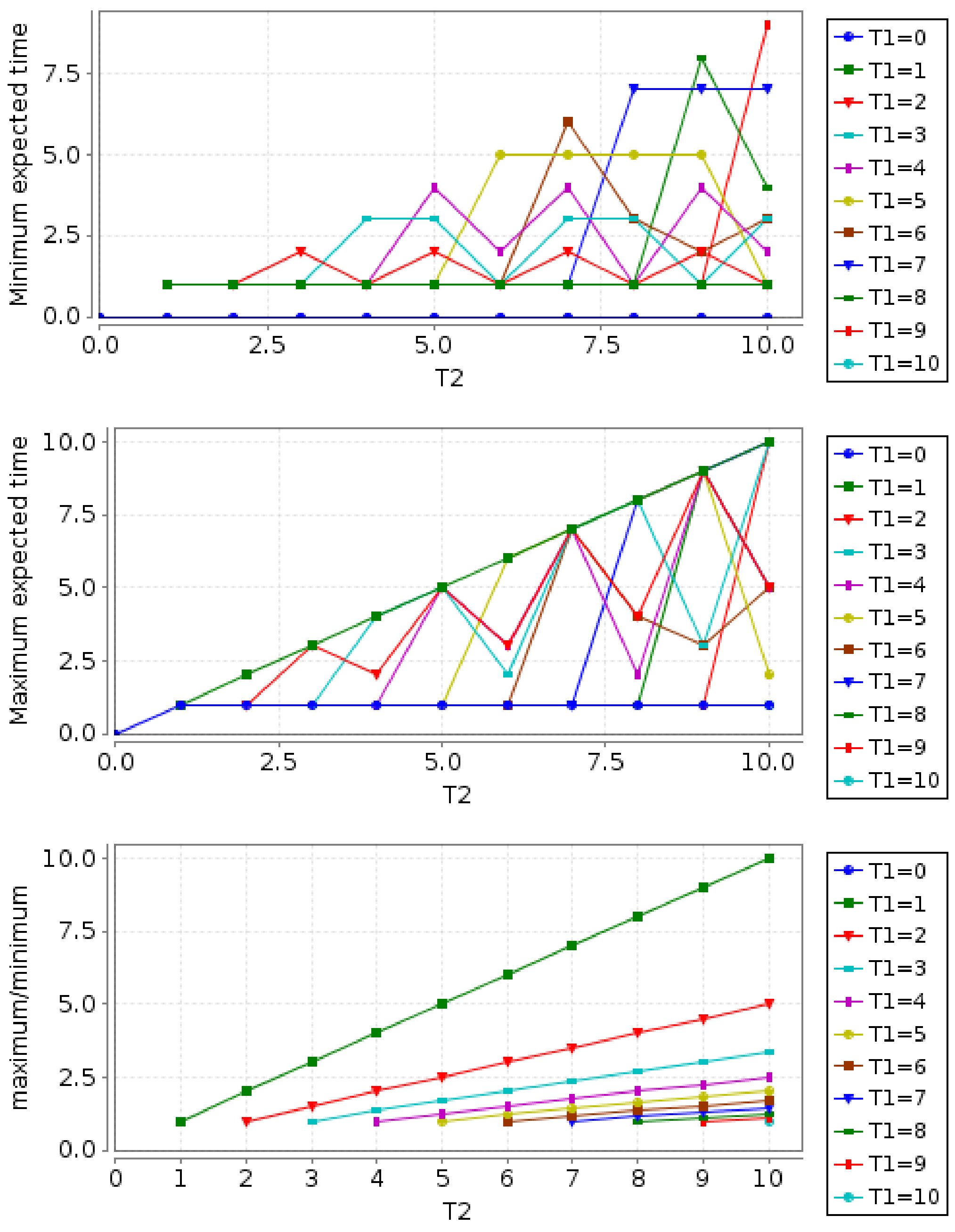


Figure 5: Accumulating Times in a PTA

with a single transition that must occur in interval $[T_1, T_2]$ with the expectation that the queries

```
Rmin=? [ F state=1 ]
Rmax=? [ F state=1 ]
```

yield T_1 and T_2, respectively. However, running the queries for various values of T_1 and T_2 yields the results depicted in Figure 5. We see that our expectation only holds in general only for $T_1 = 1$, for other values we get a curve whose values grow and shrink with periodicity T_1; as shown by the bottom diagram, the quotient of the results of the maximum/minimum values yields result T_2/T_1.

We cannot explain this behavior and have asked the authors of PRISM for clarification.

4. A Client/Server Model

Having investigated in Section 3 the various possibilities for measuring time in PRISM, our goal is now to apply these techniques to a client/server system as the one in Section 2. For this purpose, however, it does not suffice to just remember the overall number of pending requests; rather we have to remember the *identity* of each request such that we can relate the state when a request has been serviced to the previous state when this particular request has been generated.

While it would be very natural to extend the queue model such that it stores the identity of the sender of every request, the problem arises that PRISM only supports for state variables only atomic types (booleans, integers, reals) but no *arrays* which would allow to model such queues in a straight-forward way. The only way to model the request states is thus to provide a separate state variable for each request.

The most convenient way to proceed along this line is to encapsulate each state variable in a module and to utilize PRISM's feature of "module renaming": given a module

```
module M1 ... s1 ... e1 ... endmodule
```

with state variable s_1 and event e_1, we may create a copy

```
module M2 [ s1 = s2, e1 = e2 ] endmodule
```

with state variable s_2 and event e_2. By copying a module $N-1$ times, we can thus describe N states in total.

However, there is a catch: if these modules want to interact with another module M, it can only be by individual actions on these states/events, i.e., M gets structure

```
module M
  [e1] ... s1 ...
  [e2] ... s2 ...
  ...
  [eN] ... sn ...
endmodule
```

which cannot be simply created by copying but requires manual construction.

To generate a variety of N-state models, we can not just change some parameter, but we thus have to write a separate model for each specific value of N. However, to simplify our life, we might also write a script that generates the corresponding code (analogous to the PEPA model described in [2] which is generated by a script that can be downloaded from the "Examples" section of the PEPA web site [5]).

We will the following investigate this approach in two kinds of models, one where pending requests concurrently compete for access to the server, and one where pending requests are queued in the order in which they arrive, in analogy to the original queueing model. We will allow at most 5 pending client requests (i.e., $N = 5$), where each request is generated with rate 10; thus $N \cdot \lambda = 50$ which is the generation rate that was used in the queue model in Section 2.

4.1. Request Competition

In our first approach, we let clients concurrently compete for access to the server. For this purpose, we construct a CTMC model (see the code in Appendix C.1) where a client is modeled as follows:

```
module C1
  client1: [0..2] init 0;
  []           client1 = 0 -> RC : (client1' = 1);
  [request1]   client1 = 1 -> (client1' = 2);
  [response1] client1 = 2 -> (client1' = 0);
endmodule
```

The client cycles among tree states: in state 0, a request is generated (by an unnamed and thus unsynchronized action) at rate RC; in state 1, the client waits for the server to accept the request; in state 2, the client waits for the server to return a response. The corresponding actions of the server are as follows:

```
module Server
  request: [0..N] init 0;
  [request1]   request = 0 -> RS/PS: (request' = 1);
  [response1] request = 1 -> RS/(1-PS): (request' = 0);
  ...
endmodule
```

The server cycles among two states: one in which it accepts a request from some client and stores in a state variable the identity of the client; and one where it returns a response to the client whose identity was stored. The expected total time for a cycle is $1/RS$; of this a fraction PS is spent in the first state, and a fraction $1 - PS$ is spent in the second state; this determines the rates of corresponding transitions.

We use various rewards to measure the average number of clients in state 1 (waiting for the request to get accepted) and state 2 (getting serviced, i.e., waiting for a response):

```
rewards "accept"
```

```
    client1 = 1 : 1 ;
    ...
  endrewards

  rewards "service"
    client1 = 2 : 1 ;
    ...
  endrewards
```

Furthermore, we use a state reward to measure the time spent in each state:

```
  rewards "time"
    true : 1 ;
  endrewards
```

The queries

```
  R{"accept"}=? [ S ];
  R{"service"}=? [ S ];
```

then determine the average values of the state rewards, while the queries

```
  filter(avg, R{"time"}=? [ F client1=2 ], client1=1);
  filter(avg, R{"time"}=? [ F client1=0 ], client1=1);
```

compute the average time spent by a client traversing from the time a request was generated (state 1) to the time the request was accepted (state 2) respectively the response was received (state 0); here we use the new PRISM 4.0 “filter” concept which allows to state (as each third argument) the start states where a query (the second argument) shall be evaluated; for all possible start states the result are to be combined by the function denoted by the first argument (in our queries, the average of the results is computed). In Figures 6 and 7, we give the result for $N = 5$ clients with generation rate $RC = 10$, $RS \in [10, 100]$ and $PS \in [0.1, 0.5]$.

In Figure 6, we see that for lower values of RS, we have an higher utilization of the server (i.e., the average number of clients getting serviced), because request acceptance is less of a bottleneck. For large RS, we have with lower values of RP a lower number of clients waiting for acceptance of their request, as can be naturally expected. However, for small RS (where the service rate represents the bottleneck for the generation of new requests), the effect is apparently the other way round; for this we have currently no intuitive explanation.

In Figure 7, we see the average times until a message is accepted and the average time until the response is received; as expected, the difference is the reciprocal of the service rate, i.e., it is described by the formula $(1 - PS)/RS$. The core figure is therefore the average acceptance time, which corresponds to the time that in a queueing model a request has to wait in the service queue. Comparing these results with those displayed in Figures 1 and 3 for $K = 5$ and $\lambda = 50$, we see that the results derived with the new model differ considerably from the results of the queue model. In particular, the number of requests pending in the queue is much bigger than the number of clients waiting for the request of their acceptance; also the time for a request waiting in the queue is much bigger than a client waits for its request to be accepted.

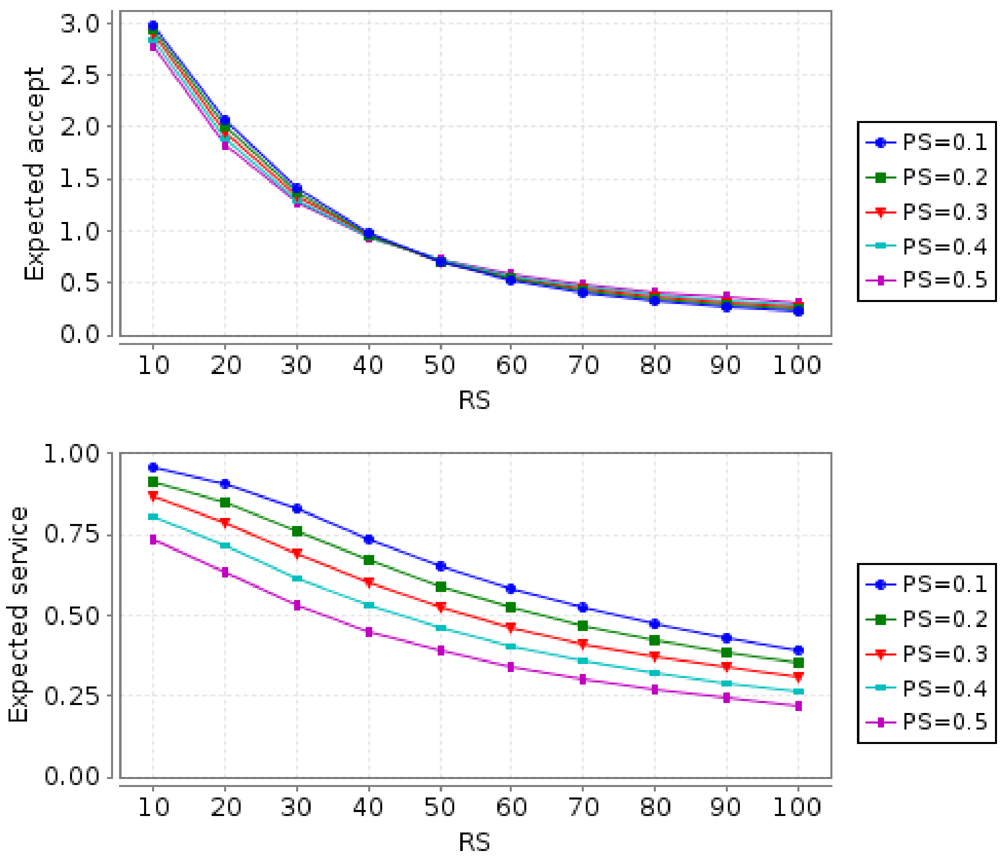


Figure 6: Request Competition (Measures)

This difference between the previously presented original queue model and the now presented client server model is a consequence of the fact that in the new model the rate with which requests are generated is not any more independent of the rate with which the responses are returned, because a client can only generate another request after it has received a response for its previous request. The state model for $N = 5$ clients is quite complex, below we show the one for $N = 2$ (where $\lambda = RC, \mu = RS, and p = PS$):

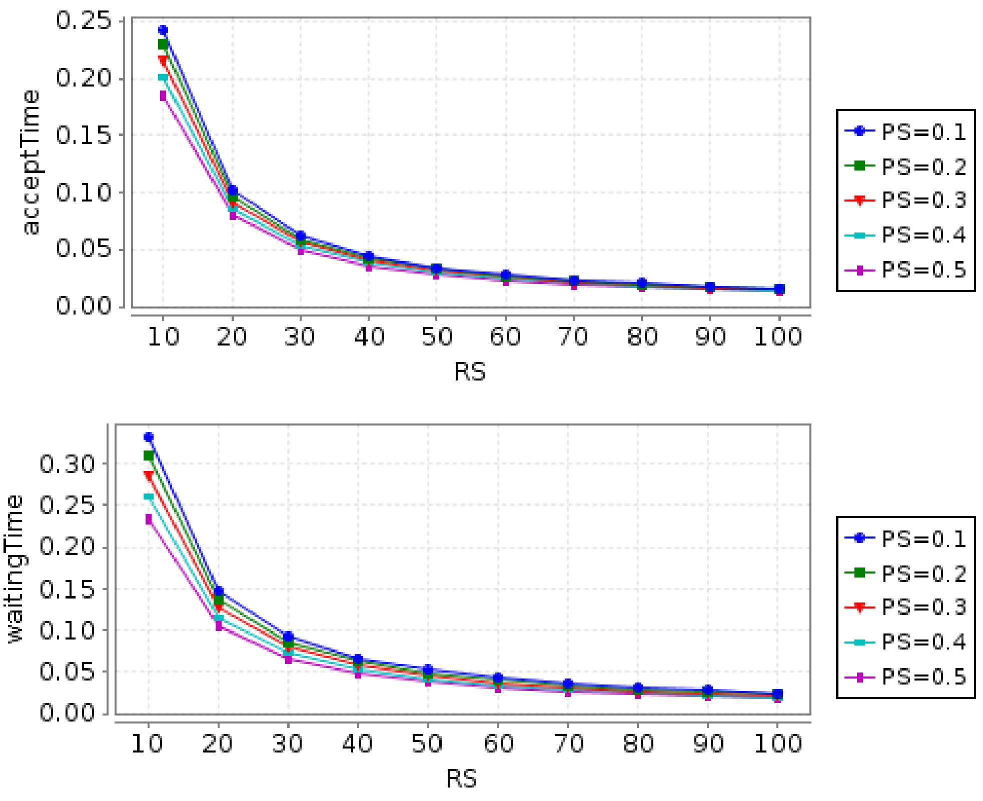
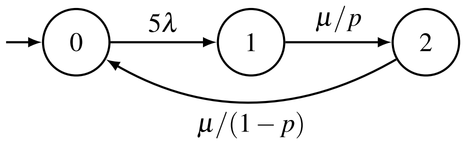


Figure 7: Request Competition (Times)

$$\begin{tikzcd}
 & & {} \arrow[d] & & \\
 & & \langle 0,0\rangle \arrow[dl, "\lambda"'] \arrow[dr, "\lambda"] & & \\
 & \langle 1,0\rangle \arrow[dl, "\mu/p"'] \arrow[dr, "\lambda"] & & \langle 0,1\rangle \arrow[dl, "\lambda"'] \arrow[dr, "\mu/p"] & \\
\langle 2,0\rangle \arrow[uurr, bend left, "\mu/(1-p)"] & & \langle 1,1\rangle \arrow[dl, "\mu/p"'] \arrow[dr, "\mu/p"] & & \langle 0,2\rangle \arrow[uull, bend right, "\mu/(1-p)"'] \\
 & \langle 2,1\rangle \arrow[uurr, bend left, "\mu/(1-p)"] & & \langle 1,2\rangle \arrow[uull, bend right, "\mu/(1-p)"'] &
\end{tikzcd}$$

Please note that the state $\langle 2,2\rangle$ where both clients wait for an answer from the server is not reachable.

We refrain from a detailed manual analysis (which would require to set up and solve a linear equation system which is exactly what PRISM does). However, to demonstrate that the request generation rate is indeed substantially reduced, we use the following simple state model where one client generates a request with rate 5λ and the server accepts the request with rate μ/p and produces a response with rate $\mu/(1-p)$:



The expected time to traverse one state cycle in this model is

$$\frac{1}{5\lambda}+\frac{p}{\mu}+\frac{1-p}{\mu}=\frac{5\lambda+\mu}{5\lambda\mu}$$

i.e., we traverse every state in the cycle with rate $\bar{\lambda}=5\lambda\mu/(5\lambda+\mu)$ (which is less than the original request rate 5λ and less than the original service rate μ).

For further illustration, Figure 8 depicts the results of an analysis of a bounded queue (see Section 2) with size $K=5$ and reduced arrival rate $\bar{\lambda}$: the expected number of pending requests, the utilization ρ, the expected time for a request to remain in the queue, and the total time for a request to be processed including the service time. We see that the values are in a similar range as those presented in Figures 6 and 7 and start to get close for large values for $\mu = RS$; however, for small values they are substantially different.

As already stated, the discussion above does not demonstrate a real analysis of the "competing requests" model but is just intended to enhance our intuition; an actual analysis would have to use the same techniques that are applied by PRISM.

4.2. Request Queueing

In this section, we investigate a model where requests are queued at the server in the order in which they arrive. Unlike the model presented in Section 2, however, we preserve in the queue the identity of the server of each request such that we can for an individual request measure the time it remains in the queue.

We model the queue by the following PRISM module (see the code in Appendix C.2):

```
// the queue (t denotes the position of the tail)
formula t = h+n < N ? h+n : h+n-N;
module Queue
  q0: [1..N] init 1; q1: [1..N] init 1; q2: [1..N] init 1;
  q3: [1..N] init 1; q4: [1..N] init 1;
  h: [0..N-1] init 0; n: [0..N] init 0;

  [request1] n < 5 & t = 0 -> 2*RS/PS : (q0' = 1) & (n' = n+1);
  [request1] n < 5 & t = 1 -> 2*RS/PS : (q1' = 1) & (n' = n+1);
  [request1] n < 5 & t = 2 -> 2*RS/PS : (q2' = 1) & (n' = n+1);
  [request1] n < 5 & t = 3 -> 2*RS/PS : (q3' = 1) & (n' = n+1);
```

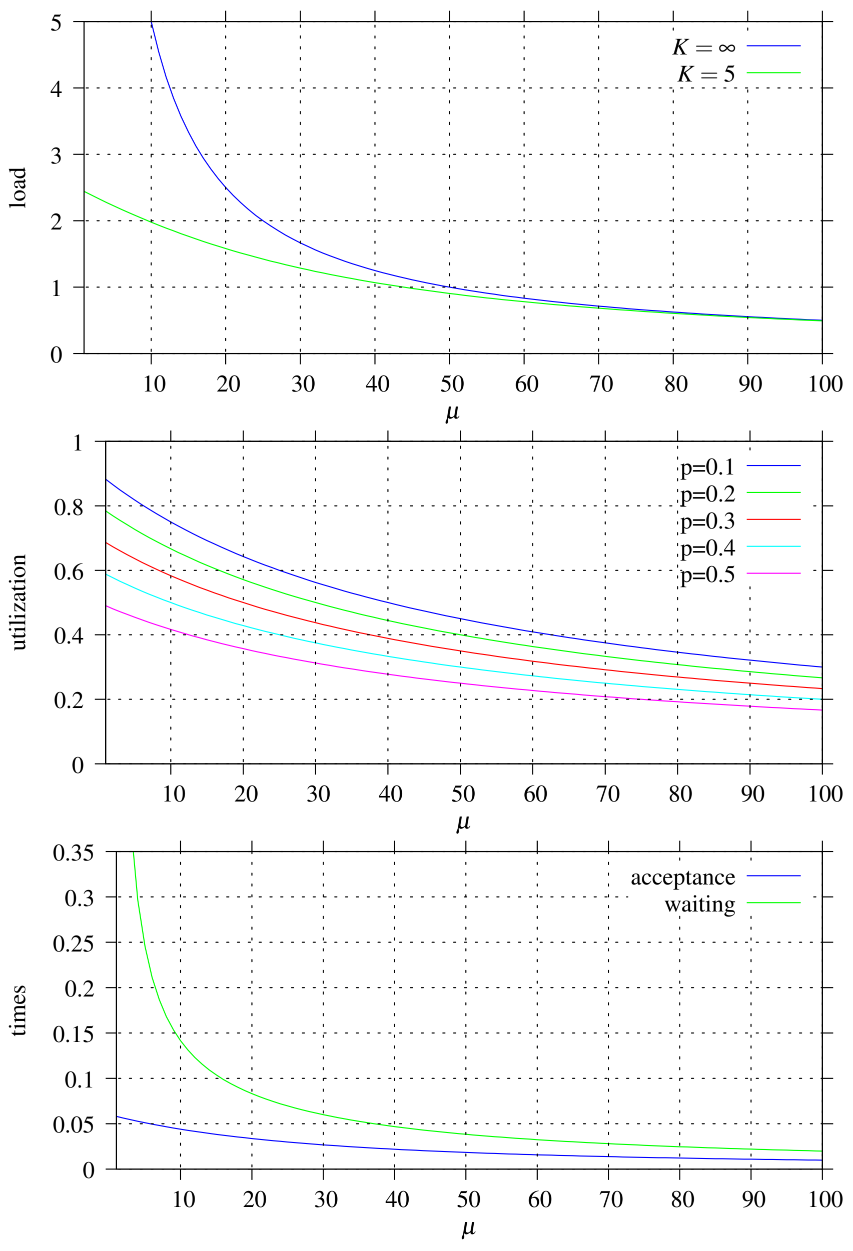


Figure 8: Bounded Queue with Reduced Arrival Rate

```
    [request1] n < 5 & t = 4 -> 2*RS/PS : (q4' = 1) & (n' = n+1);
    [forward1] n > 0 & h = 0 & q0 = 1 -> (h' = 1) & (n' = n-1);
    [forward1] n > 0 & h = 1 & q1 = 1 -> (h' = 2) & (n' = n-1);
    [forward1] n > 0 & h = 2 & q2 = 1 -> (h' = 3) & (n' = n-1);
    [forward1] n > 0 & h = 3 & q3 = 1 -> (h' = 4) & (n' = n-1);
    [forward1] n > 0 & h = 4 & q4 = 1 -> (h' = 0) & (n' = n-1);

    // [requestI] and [forwardI] for I=2..5
    ...
  endmodule
```

The module has 5 state variables $q_0, \ldots, q_4$ that represent the 5 positions in the queue; index h represents the position of the head of the queue, counter n represents the number of elements in the queue. The position where the next element is to be inserted is thus $(h+n)$ mod 5 which we denote by the "virtual" variable t introduced by the PRISM "formula" construct (we do not use a real variable in order to reduce the size of the state space which will become very large).

For each value of t, we have now a transition "request1" to receive a request from client 1 into position t; likewise, we have for each value of h a transition "forward1" to forward the request from client 1 stored at position h to the server; for a single client we thus have $2N$ transitions, for N clients we have $2N^2$ transitions. Since the module has $N+2$ variables with N values each, the size of the state space is N^{N+2}, i.e., grows super-exponentially with the number of clients.

We assign to the "request" transitions rate $2 \cdot RS/PS$, i.e., the double of the original rate, because we want to split the time previously assigned to "request" now between "request" and "forward". The modules for the client and the server remain as in the previous section except that the "request" transitions in the server are translated into corresponding "forward" transitions that synchronize with the queue module rather than with the client modules. As explained above, these transitions now also receive rate $2 \cdot RS/PS$.

We use the reward structures

```
rewards "waiting"
  client1 = 1 | client1 = 2 : 1 ;
  client2 = 1 | client2 = 2 : 1 ;
  client3 = 1 | client3 = 2 : 1 ;
  client4 = 1 | client4 = 2 : 1 ;
  client5 = 1 | client5 = 2 : 1 ;
endrewards

rewards "utilization"
  request != 0 : 1;
endrewards

rewards "time"
  true : 1 ;
endrewards
```

and the queries

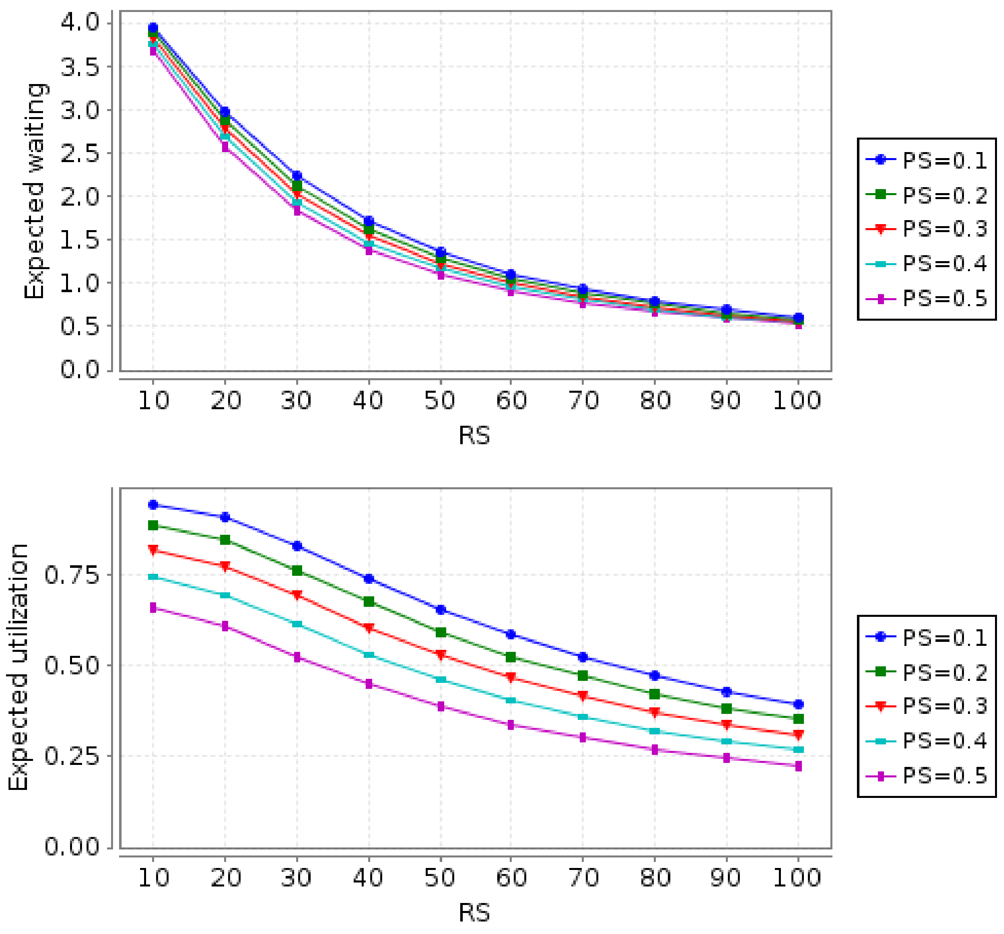


Figure 9: Request Queueing (Measures)

```
"waitingNumber": R{"waitingNumber"}=? [ S ];
"utilization": R{"utilization"}=? [ S ];
"waitingTime":
  filter(avg, R{"time"}=? [ F client1=0 ], client1=1);
"queueTime": "waitingTime"-(1-PS)/RS;
```

to determine the average number of waiting clients, the utilization of the server, the time that a client spends in waiting (from the generation of a request to the receipt of a response) and this time without the actual service time (since the execution of the server is now decoupled from the execution of the client it does not make sense to measure just the time for the transition from state 1 to state 2).

Checking this model pushes PRISM not too far yet: the system has $S = 1,273,200$ states and $T = 6,005,200$ transitions. The model is constructed in about 2 seconds (on a machine with

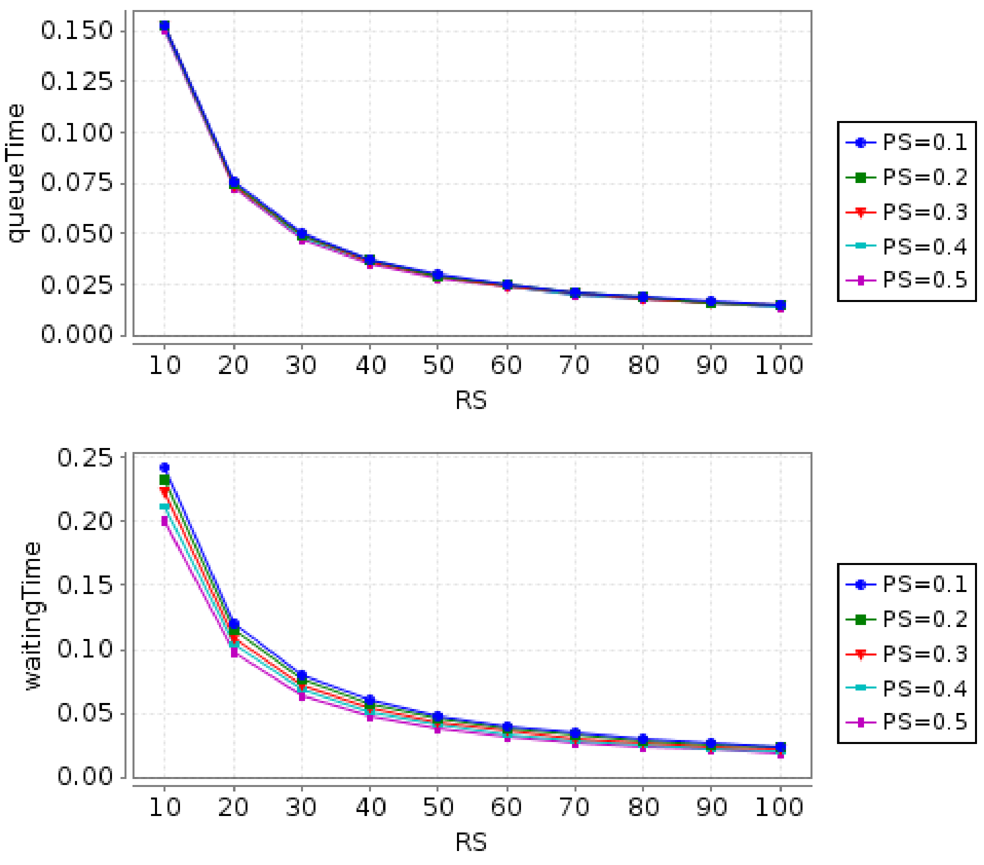


Figure 10: Request Queueing (Times)

an Intel Core i7-2670 CPU with 2.2GHz clock frequency) and takes about 30 MB of space. For checking the model we choose the fast “Sparse” engine (the default “Hybrid” engine is much too slow); the equation system is therefore represented by a sparse $S \times S$ matrix with T non-zero entries. For solving this system we choose the “JOR” (Jacobi Over-Relaxation) solver (the default “Jacobi” solver does mostly not converge, “Gauss-Seidel” does not converge for the steady state analysis); a model check can be then performed in a bit less than 5 seconds.

Comparing the results depicted in Figures 9 and 10 with those presented in Figures 6 and 7 for the “competing requests” model, we see that

- the number of clients waiting in the queue is now higher than the number of clients waiting for acceptance before,
- the utilization of the server is now lower than before,
- but the overall waiting time is now somewhat smaller, because
- the time spent in the queue is smaller than the previously determined time for acceptance.

As for the last two effects, we do not have any good explanation, our expectation would have to be otherwise. Further investigations are needed to explain these results.

Of course, all the results are considerably different from those for the queue model presented in Section 2, because the request arrival rate is now much smaller. We will therefore now turn to a model that more directly corresponds to the initially presented queue model.

4.3. The Queue-Based Model Revisited

We are now going to model a system where the clients do not wait for answers produced by the server but just generate requests at a constant rate:

```
module C1
  client1: [0..1] init 0;
  []           client1 = 0 -> RC : (client1′ = 1);
  [request1]   client1 = 1 -> (client1′ = 0);
endmodule
```

The queue between clients and server rejects messages if the number of pending requests is bigger than the number of clients ($K = N = 5$):

```
formula t = h+n < N ? h+n : h+n-N;
module Queue
  q0: [1..N] init 1; q1: [1..N] init 1; q2: [1..N] init 1;
  q3: [1..N] init 1; q4: [1..N] init 1;
  h: [0..N-1] init 0; n: [0..N]   init 0;
  accepted: bool init true;

  // requests from client 1
  [request1] n = 5 -> 2*RS/PS: (accepted′ = false);
  [request1] n < 5 & t = 0 -> 2*RS/PS:
    (q0′ = 1) & (n′ = n+1) & (accepted′ = true);
    ...
  [forward1] n > 0 & h = 0 & q0 = 1 -> (h′ = 1) & (n′ = n-1);
    ...

  // requests from the other clients
  ...
endmodule
```

We associate to this model reward structures

```
rewards "rejected"
  !accepted : 1;
endrewards
rewards "waiting"
  true : n;
endrewards
rewards "time"
```

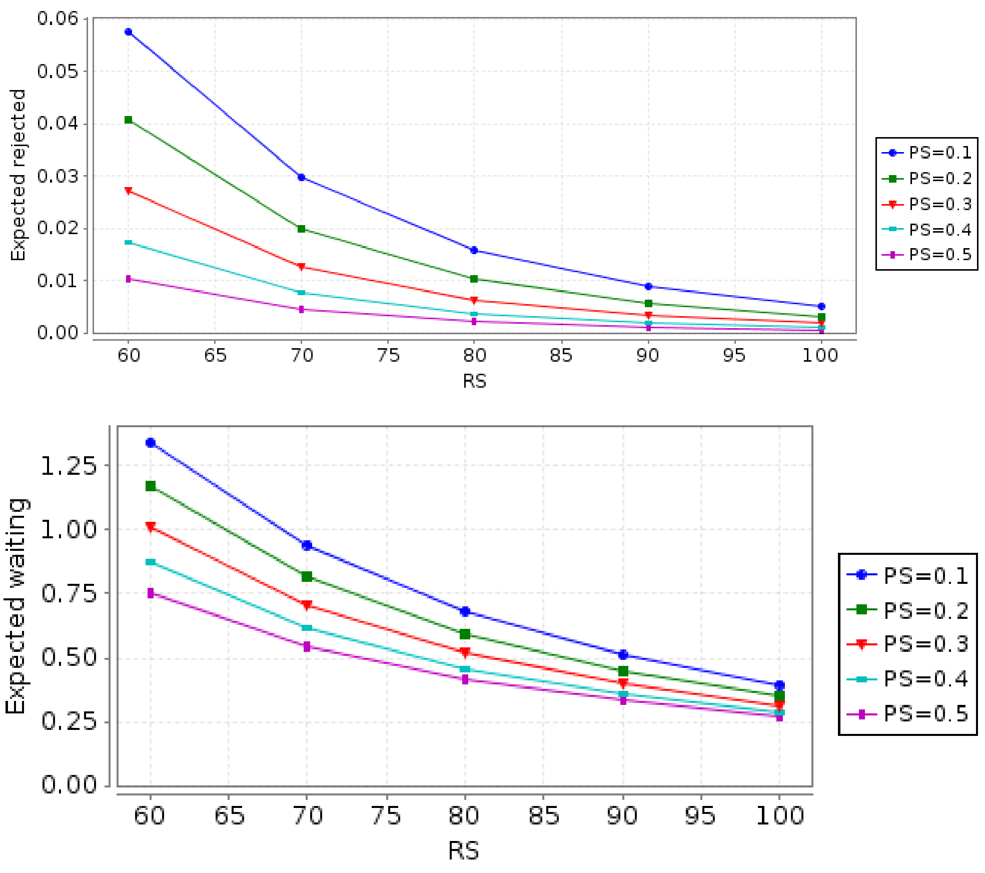


Figure 11: The Queue-Based Model Revisited (Measures)

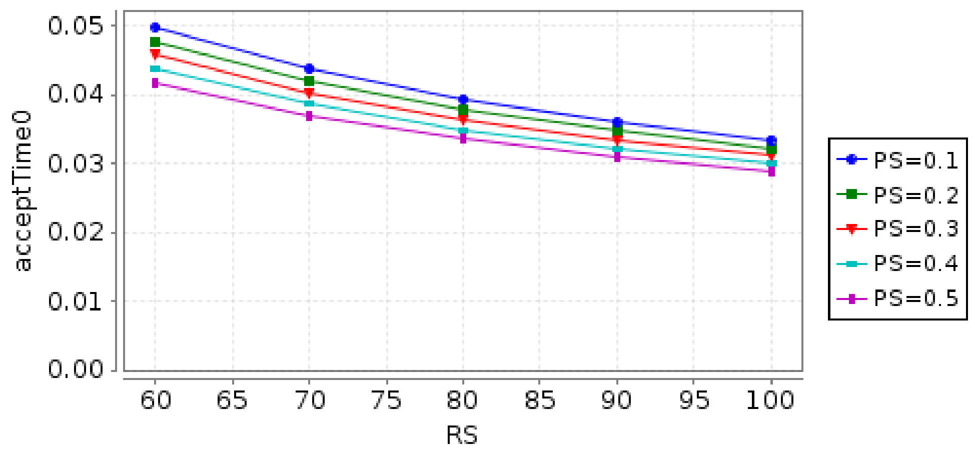
```
  true : 1 ;
endrewards
```

such that by the queries

```
"rejected":  R{"rejected"}=? [ S ];
"waiting":  R{"rejected"}=? [ S ];
"acceptTime0":
  filter(avg, R{"time"}=? [ F request=1 ], client1=1);
"waitingTime0": "acceptTime0"+(1-PS)/RS;
```

we can query the probability of a request to be rejected, the average number of requests in the queue the time for acceptance of request by the server, and the overall time for processing a message from the time the request was generated to the time when the response was returned.

The model is now much bigger than the one in the previous section, it comprises 16 million states and 95 million transitions, is constructed in about 10 seconds and takes about 600 MB of



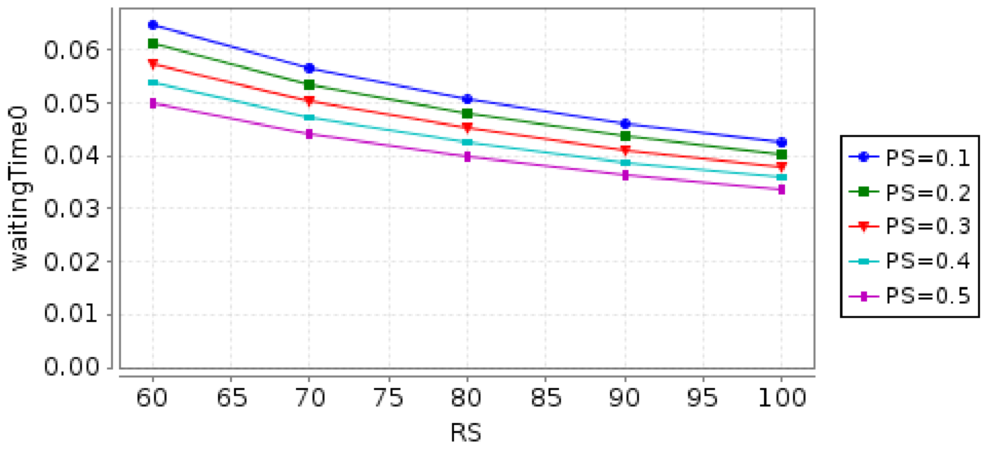


Figure 12: The Queue-Based Model Revisited (Times)

memory. For model checking, we again use the "Sparse" engine and the "JOR" solver, which solves the equation system (depending on the parameter values) in 2–5 minutes; to compute all data points for an experiment takes a considerable amount of time. We thus have started to encounter the well-known "state space explosion" problem of model checking, which shows that the practical limit for the use of PRISM in experiments with changing parameters is in the order of 10^7 to 10^8. To reduce the size of the model, we may e.g. drop the boolean variable *accepted* which was only introduced to measure rejection probabilities; since the size of the state space is now halved, the time for solving the equation system is approximately reduced by a factor of 4 (it is now every bit in the state model that counts)!

Figures 11 and 12 depict the results of the analysis. We can compare this with those presented for the bounded queue model with $K = 5$ in Section 2 and see, that for a small value of PS the new results are similar (but not identical) the old ones with considerable differences for small values of $\mu = RS$ (e.g., the rejection rate for $\mu = 60$ was originally 0.1; now we have for $RS = 60$

and $PS = 0.1$ rejection rate 0.05).

While from a pragmatical point of view the differences may not be considered as too important, we would nevertheless like to investigate the core reason for the differences in more detail. Comparing the original queue model with the current one, we see that in the original model there were just two state transitions for handling a message ("send" and "receive") with rates λ and μ while we have now four transitions (the unnamed client transition to generate a request, then "request1", then "forward1", and finally the unnamed transition of the server to process a request) with rates RS, $2RS/PS$, $2RS/PS$, and $RS/(1-PS)$. Thus the new model may be in some respect quite similar to the old one, but it is definitely not the same. In the following, we will therefore investigate whether it is possible to construct a model with explicit time measurement that is closer to the original one.

4.4. The Queue-Based Model Revisited Again

We are now going to construct a queue-based model with explicit time measurement where handling a message requires only two transitions as in the original model presented in Section 2. Furthermore, we would like to minimize the state space as much as possible such that model checking time becomes small again.

In our new model the client is modeled as

```
module C1
  [request1] true -> RC : true;
  [reject1]  true -> RC : true;
endmodule
```

i.e., it generates at rate RC that is either accepted or rejected; we will make sure that in every state only one of these transitions is enabled such that the overall rate of request generation is the same as in the original model.

The queue is now modeled as

```
formula t = h+n < N ? h+n : h+n-N;
module Queue
  q0: [1..N] init 1; q1: [1..N] init 1; q2: [1..N] init 1;
  q3: [1..N] init 1; q4: [1..N] init 1;
  h: [0..N-1] init 0; n: [0..N] init 0; r: [0..N] init 0;

  // accept/reject request from client 1
  [reject1] n = 5 -> true;
  [request1] n < 5 & t = 0 -> (q0' = 1) & (n' = n+1) & (r' = 1);
  [request1] n < 5 & t = 1 -> (q1' = 1) & (n' = n+1) & (r' = 1);
  [request1] n < 5 & t = 2 -> (q2' = 1) & (n' = n+1) & (r' = 1);
  [request1] n < 5 & t = 3 -> (q3' = 1) & (n' = n+1) & (r' = 1);
  [request1] n < 5 & t = 4 -> (q4' = 1) & (n' = n+1) & (r' = 1);
  [forward1] n > 0 & h = 0 & q0 = 1 -> (h' = 1) & (n' = n-1);
  [forward1] n > 0 & h = 1 & q1 = 1 -> (h' = 2) & (n' = n-1);
  [forward1] n > 0 & h = 2 & q2 = 1 -> (h' = 3) & (n' = n-1);
  [forward1] n > 0 & h = 3 & q3 = 1 -> (h' = 4) & (n' = n-1);
```

```
    [forward1] n > 0 & h = 4 & q4 = 1 -> (h' = 0) & (n' = n-1);

    // the same for the other clients
    ...
  endmodule
```

The queue is extended in this model to set the state variable *r* to the number of the client whose request was received last. On the other side, the rejection of a message is now not handled any more by a separate state variable but by a separate transition. Please note that inside the module all transitions have rate 1 (the default value) such that the overall rate of the synchronized transition is now the one with which the partner component is ready to engage; in particular, requests are accepted/rejected with rate *RC* and (as we will see below) forwarded with rate *RS*.

The server model now becomes

```
  module Server
    request: [0..N] init 0;
    [forward1] true -> RS: (request' = 1);
    [forward2] true -> RS: (request' = 2);
    [forward3] true -> RS: (request' = 3);
    [forward4] true -> RS: (request' = 4);
    [forward5] true -> RS: (request' = 5);
  endmodule
```

i.e., it simply accepts and processes requests with rate *RC* and sets its state variable *request* to the number of the client whose request was processed last.

We assign to the model the reward structures

```
  rewards "rejected"
    [reject1] true : 1;
    ...
  endrewards
  rewards "accepted"
    [request1] true : 1;
    ...
  endrewards
  rewards "waiting"
    true : n;
  endrewards
  rewards "time"
    true : 1 ;
  endrewards
```

that assign to every rejection of a message value 1, to every acceptance of a message value 1, to every state the number of requests in the queue (multiplied with the time spent in that state), and to every state the time spent in that state, respectively. Please note that, in contrast to our previous model, we do not have a state variable any more to denote acceptance/rejection of a

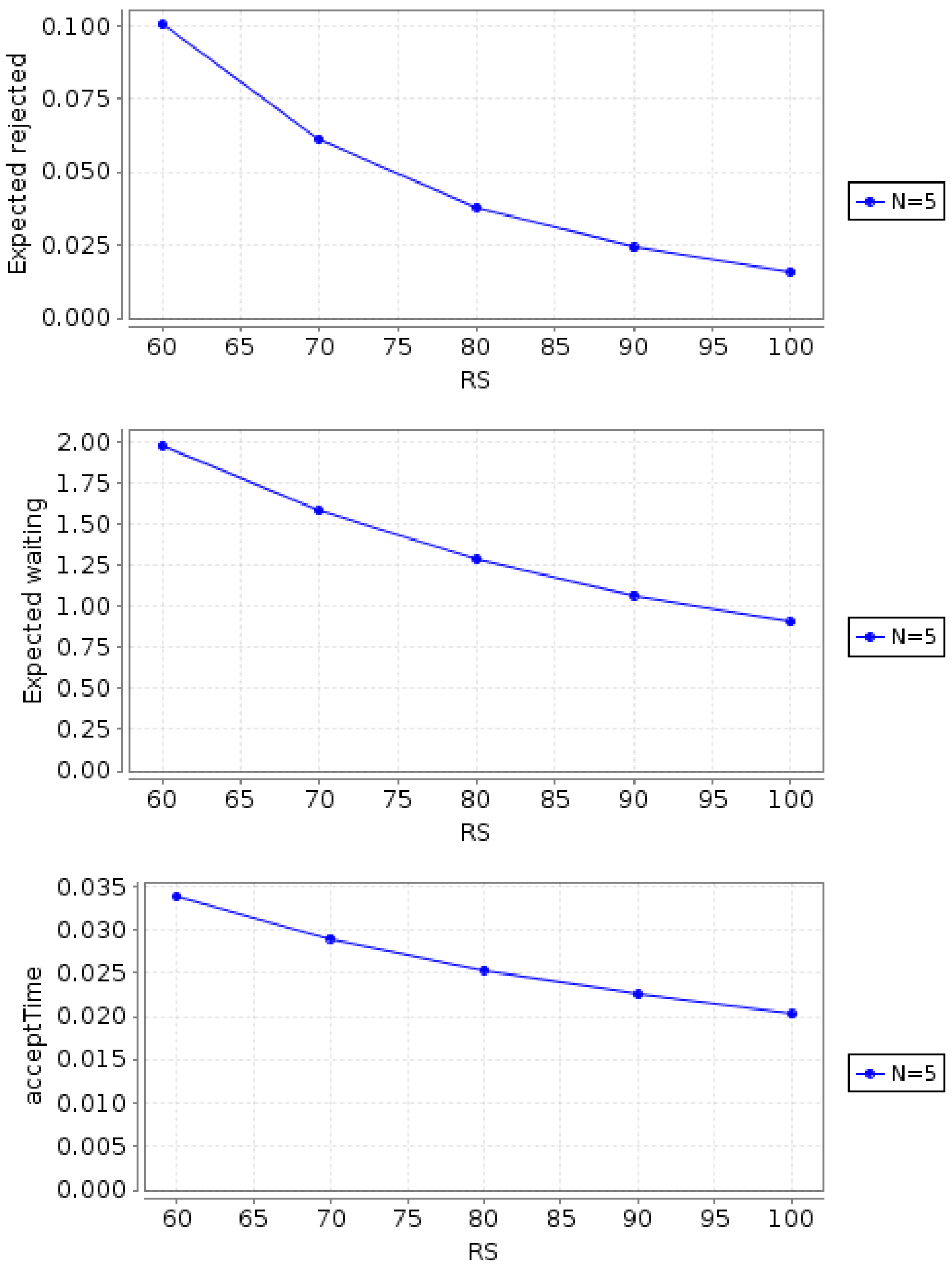


Figure 13: The Queue-Based Model Revisited Again

message but count by transition rewards the average number of messages accepted and rejected. By the CSL query

```
"rejectedN": R{"rejected"}=? [ S ];
"acceptedN": R{"accepted"}=? [ S ];
"rejected":  1/(1+"acceptedN"/"rejectedN");
```

we can calculate the probability of a message rejected as $a/(a+r) = 1/(1+r/a)$ where a is the average number of accepted messages and r is the average number of rejected messages. The calculation of the rejection rate thus requires now two steady-state reward computations rather than one; however, since the model is now much smaller than the previous one, this computation is still faster than before. By the queries

```
waiting":      R{"waiting"}=? [ S ];
"acceptTime": filter(avg, R{"time"}=? [F request=1], r=1);
```

we can compute tha average number of requests pending in the queue and the average time it takes from the generation of a message from client 1 (when the queue sets variable r to 1) to the time the message is accepted by the server (when the server sets variable *request* to 1).

The new model has 160156 states and 620310 transitions; it can be constructed in about 7 seconds and takes about 5 MB of memory; solving the equation system takes 7–120 seconds, depending on the query and the parameter values. The results of the analysis are depicted in Figure 13. Comparing them with those presented in Section 2, we see that the new model indeed has the same rejection probability and the same number of requests pending in the queue as the original one; however, the time from the point when a message is put into the queue to the time when it leaves the queue is for small $\mu = RS$ slightly smaller than in the original model (for $\mu = RS = 60$, it is determined as 0.044 in the original model but as 0.034 in our new model); for large $\mu = RS$, it is slightly higher (for $\mu = RS = 60$, we have in the old model 0.018 but 0.020 in the new model). So apparently there is a small discrepancy between the models with respect to timing which we are not yet able to explain.

5. Conclusions

From the work reported in this paper, our general experience is that PRISM is a powerful and versatile tool to model and analyze the performance of computing systems, also with respect to computation times. PRISM 4.0 has introduced a number of new features that help us in this respect, in particular we have used the possibility of nested reward computations (which allows us to avoid the use of external tools to compute combined measures) and the new style of filters (which allow to analyze all the paths from a certain set of start states to a certain set of end states). The new support for "Probabilistic Timed Automata" (PTAs) extends the expressiveness of "Markov Decision Processes" (MDPs); however since both models are based on non-deterministic (rather than probabilistic/stochastic) choice among multiple execution paths, we have found them less attractive for our purposes. Also we have encountered some aspects of reward computation in PTAs which we do not yet understand (we have asked the PRISM authors for clarification). In general, we find the use of continuous time models (CTMCs) preferable over that of discrete time models (DTMCs) since the use of execution rates rather than probabilities seems to us much more natural for performance modeling.

As for the performance modeling of client/server systems, we see that there is a considerable difference between modeling a system with a fixed number of clients submitting requests and waiting for responses and a system with a fixed rate of incoming requests that are generated independently from the responses. If a fixed number of clients waits for responses, the system can be easily modeled as a finite state system with a relatively small number of states that make it amenable for efficient analysis.

Also if requests are generated with a fixed rate independently of the responses, the system can be modeled with by a bounded queue with a finite number of states, but this requires PRISM modules with N^K states and $2NK$ transitions (where N is the number of clients and K is the length of the queue); writing such modules by hand is tedious and error-prone, so automatic generation by a script would be preferable. The system may be then analyzed with PRISM in the same way as in the other case; the results are quite close to those predicated by classical queueing theory (but there are small differences that still remain to be explained).

However, due to the size of the state space this kind of analysis is effective only for small systems. For large systems our earlier approach of modeling only the number of elements in a queue and applying Little's Law to determine response times is much more effective. For such situation, the detailed analysis of the queue model performed in this paper should be helpful.

As for the performance of PRISM 4.0, we see that with today's processing power and memory capacities models whose state space sizes are in the order of 10^7 (where checking time is in the order of a few minutes) can be reasonably well dealt with (i.e., we can comfortably study models with varying parameters; for an individual analysis, also a state space size in the order of 10^8 may be acceptable). Here the choice of the checking engine ("Sparse" is much faster than the default engine "Hybrid") and of the linear equation solver (the default "Jacobi" solver did mostly not converge, "Gauss-Seidel" did not converge for steady state reward computations, all in all "JOR" performed best) is critical.

Having gained more knowledge about the capabilities of PRISM 4.0, in particular for but not restricted to the purpose of time measurement, we plan to apply our insights to modeling various systems in the area of mobile communication and compare them with the results derived from other models and supporting software tools (e.g,, the performance process algebra PEPA [5]). Also the use of the forth-coming PRISM 4.1 (which will support a new explicit-state model checking engine) will become interesting.

References

[1] Tamas Berczes et al. "Evaluating a Probabilistic Model Checker for Modeling and Analyzing Retrial Queueing Systems". In: *Annales Mathematicae et Informaticae* 37 (2010), pp. 51–75.

[2] J.T. Bradley et al. "Derivation of Passage-Time Densities in PEPA models using ipc: The Imperial PEPA Compiler". In: *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003 (MASCOTS 2003), 11th IEEE/ACM International Symposium*. Oct. 2003, pp. 344–351.

[3] Tamás Bérczes et al. “Analyzing a Proxy Cache Server Performance Model with the Probabilistic Model Checker PRISM”. In: *Automated Specification and Verification of Web Systems (WWV’09)* (July 17, 2009). Ed. by Demis Ballis and Temur Kutsia. 2009. URL: http://www.risc.jku.at/publications/download/risc_3852/WWV-2009-Proceedings.pdf.

[4] Robert B. Cooper. *Introduction to Queueing Theory*. 2nd. North Holland, 1981.

[5] Jane Hillston and Stephen Gilmore, eds. *PEPA Process Performance Evaluation Algebra*. Laboratory for Foundations of Computer Science, The University of Edinburgh, UK. 2013. URL: http://www.dcs.ed.ac.uk/pepa.

[6] 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.

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

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

[9] Katinka Wolter, ed. *Formal Methods and Stochastic Models for Performance Evaluation. Fourth European Performance Engineering Workshop, EPEW 2007*. Lecture Notes in Computer Science 4748. 2007.

A. A Queue-Based Model of a Client/Server System

```
// -------------------------------------------------------------------
// Queue.sm
// A system with a client and a server connected by a queue.
//
// 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
// -------------------------------------------------------------------

// rates
const double lambda; // arrival rate
const double mu;     // service rate

// bounds
const int K; // queue capacity

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

// generate requests at rate lambda
module Client
  accepted : bool init false;
  [send] true -> lambda : (accepted' = waiting < K);
endmodule

// bounded queue of size K
module Queue
  waiting : [0..K] init 0;
  [send]    waiting < K -> (waiting' = waiting+1);
  [send]    waiting = K -> true;
  [receive] waiting > 0 -> (waiting' = waiting-1);
endmodule

// process rates at rate mu
module Server
  [receive] true -> mu : true;
endmodule

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

rewards "waiting"
  true : waiting ;
endrewards

rewards "idle"
  waiting = 0 : 1;
endrewards

rewards "rejected"
  !accepted : 1;
endrewards


// -------------------------------------------------------------------
// Queue.csl
// -------------------------------------------------------------------

// the average number of waiting/idle/rejected states
"waiting":  R{"waiting"}=? [ S ];
"idle":     R{"idle"}=? [ S ];
"rejected": R{"rejected"}=? [ S ];

// Little's Law for M/M/1
"time0": "waiting"/lambda;

// Little's Law for M/M/1/K
"time1": "waiting"/(lambda*(1-"rejected"));
"time2": "waiting"/(lambda*(1-"rejected"))+(1/mu);
```

B. Approaches to Explicit Time Measurement

B.1. A Digital Clock with State

```
// -------------------------------------------------------------------
// Clock.sm
// A system with two concurrent components and a clock
// that counts the number of ticks.
//
// 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

const int N = 50;  // maximum number of clock cycles
const int M = 10;  // number of clock cycles per time unit

module M1
  state1: [0..1] init 0;
  [] state1 = 0 -> (state1' = 1);
endmodule

module M2
  state2: [0..1] init 0;
  [] state2 = 0 -> (state2' = 1);
endmodule

module Clock
  c: [0..N] init 0;
  [] true -> M : (c' = min(c+1,N));
endmodule


// -------------------------------------------------------------------
// Clock.csl
// -------------------------------------------------------------------

const int T;

// the probability that c remains below T until the state transition
P=? [ c < T U state1 = 1 ];
```

B.2. A Digital Clock without State

```
// -------------------------------------------------------------------
// Clock2.sm
// A system with two concurrent components and a clock that ticks.
//
// 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

const int M = 10;  // number of clock cycles per time unit

module M1
  state1: [0..1] init 0;
  [] state1 = 0 -> (state1' = 1);
endmodule

module M2
  state2: [0..1] init 0;
  [] state2 = 0 -> (state2' = 1);
endmodule

module Clock
  [tick] true -> M : true;
endmodule

rewards
  [tick] true : 1/M;
endrewards

// -------------------------------------------------------------------
// Clock2.csl
// -------------------------------------------------------------------

// the time until the state transition occurs
R=? [ F state1 = 1 ];
```

B.3. No Digital Clock

```
// -------------------------------------------------------------------
// NoClock.sm
// A system with two concurrent components and no clock.
//
// 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

module M1
  state1: [0..1] init 0;
  [step1] state1 = 0 -> (state1' = 1);
endmodule

module M2
  state2: [0..1] init 0;
  [step2] state2 = 0 -> (state2' = 1);
endmodule

rewards "time1"
```

```
  [step1] true : 1;
  [step2] true : 1;
endrewards

rewards "time2"
  [step1] state2 = 0 : 0.5;
  [step1] state2 = 1 : 1;
  [step2] state1 = 0 : 0.5;
  [step2] state1 = 1 : 1;
endrewards

rewards "time3"
  true : 1;
endrewards

// ------------------------------------------------------------------
// NoClock.csl
// ------------------------------------------------------------------

// the time until we the desired state (measured from transitions)
R{"time1"}=? [ F state1 = 1 ];

// the time until we the desired state (measured from transitions
// with refined reward structure)
R{"time2"}=? [ F state1 = 1 ];

// the time until we the desired state (measured from states)
R{"time3"}=? [ F state1 = 1 ];
```

B.4. A Digital Clock without State (DTMC)

```
// ------------------------------------------------------------------
// Clock2.pm
// A system with two concurrent components and a clock that ticks.
//
// 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
// ------------------------------------------------------------------

// discrete time Markov chain
dtmc

module M1
  state1: [0..1] init 0;
  [] state1 = 0 -> (state1' = 1);
endmodule
```

```
module M2
  state2: [0..1] init 0;
  [] state2 = 0 -> (state2' = 1);
endmodule

module Clock
  [tick] true -> true;
endmodule

rewards
  [tick] state1 = 0 & state2 = 0 : 0.5/0.5;
  [tick] state1 = 0 & state2 = 1 : 1/1;
  [tick] state1 = 1 & state2 = 0 : 1/1;
endrewards

// -------------------------------------------------------------------
// Clock2.pctl
// -------------------------------------------------------------------

// the time until the state transition occurs
R=? [ F state1 = 1 ];
```

B.5. No Digital Clock (MDP)

```
// -------------------------------------------------------------------
// NoClock.pm
// A system with two concurrent components and no clock
// (discrete time model).
//
// 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
// -------------------------------------------------------------------

// discrete time Markov chain
mdp

module M1
  state1: [0..1] init 0;
  [step1] state1 = 0 -> (state1' = 1);
endmodule

module M2
  state2: [0..1] init 0;
  [step2] state2 = 0 -> (state2' = 1);
endmodule

rewards "time1"
```

```
  [step1] true : 1;
  [step2] true : 1;
endrewards

rewards "time2"
  [step1] state2 = 0 : 0.5;
  [step1] state2 = 1 : 1;
  [step2] state1 = 0 : 0.5;
  [step2] state1 = 1 : 1;
endrewards

rewards "time3"
  true : 1;
endrewards

rewards "time4"
  state1 = 0 & state2 = 0 : 0.5;
  state1 = 1 & state2 = 0 : 0.5;
  state1 = 0 & state2 = 1 : 1;
  state1 = 1 & state1 = 1 : 1;
endrewards

// -------------------------------------------------------------------
// NoClock.pctl
// -------------------------------------------------------------------

// the time until we the desired state (measured from transitions)
R{"time1"}=? [ F state1 = 1 ];

// the time until we the desired state (measured from transitions
// with refined reward structure)
R{"time2"}=? [ F state1 = 1 ];

// the time until we the desired state (measured from states)
R{"time3"}=? [ F state1 = 1 ];

// the time until we the desired state (measured from states)
R{"time4"}=? [ F state1 = 1 ];
```

B.6. No Digital Clock (PTA)

```
// -------------------------------------------------------------------
// NoClockPTA.nm
// A system with two concurrent components and no clock
// (PTA model).
//
// 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
// ------------------------------------------------------------------

// probabilistic timed automata
pta

const int T1 = 80;
const int T2 = 120;

module M1
  state1: [0..1] init 0;
  c1: clock;
  invariant state1 = 0 => c1 <= T2 endinvariant
  [step1] state1 = 0 & T1 <= c1 -> (state1' = 1);
endmodule

module M2
  state2: [0..1] init 0;
  c2: clock;
  invariant state2 = 0 => c2 <= T2 endinvariant
  [step2] state2 = 0 & T1 <= c2 -> (state2' = 1);
endmodule

rewards "time1"
  [step1] true : 100;
  [step2] true : 100;
endrewards

rewards "time2"
  [step1] state2 = 0 : 50;
  [step1] state2 = 1 : 100;
  [step2] state1 = 0 : 50;
  [step2] state1 = 1 : 100;
endrewards

rewards "time3"
  true : 100;
endrewards

rewards "time4"
  true : 1;
endrewards

// ------------------------------------------------------------------
// NoClockPTA.pctl
// ------------------------------------------------------------------

// the time until we the desired state (measured from transitions)
R{"time1"}min=? [ F state1 = 1 ];
R{"time1"}max=? [ F state1 = 1 ];
```

```
// the time until we the desired state (measured from transitions
// with refined reward structure)
R{"time2"}min=? [ F state1 = 1 ];
R{"time2"}max=? [ F state1 = 1 ];

// the time until we the desired state (measured from states)
R{"time3"}min=? [ F state1 = 1 ];
R{"time3"}max=? [ F state1 = 1 ];

// the time until we the desired state (measured from states)
R{"time4"}min=? [ F state1 = 1 ];
R{"time4"}max=? [ F state1 = 1 ];
```

C. A Client/Server Model

C.1. Request Competition

```
// -----------------------------------------------------------------
// ClientServer1.sm
// A system with multiple clients and one server where
// the clients concurrently compete for access to the server.
//
// 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
// -----------------------------------------------------------------

const int N = 5; // number of clients

const double RC; // rate with which a client generates a request
const double RS; // rate with which the server handles a request
const double PS; // fraction of time server spends in acceptance

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

// the first client
module C1
  client1: [0..2] init 0;
  []           client1 = 0 -> RC : (client1' = 1);
  [request1]   client1 = 1 -> (client1' = 2);
  [response1] client1 = 2 -> (client1' = 0);
endmodule

// the other clients
module C2 = C1
  [ client1 = client2, request1 = request2, response1 = response2 ]
endmodule
module C3 = C1
  [ client1 = client3, request1 = request3, response1 = response3 ]
endmodule
module C4 = C1
  [ client1 = client4, request1 = request4, response1 = response4 ]
endmodule
module C5 = C1
  [ client1 = client5, request1 = request5, response1 = response5 ]
endmodule

// the server
module Server
  request: [0..N] init 0;
  [request1]   request = 0 -> RS/PS: (request' = 1);
  [request2]   request = 0 -> RS/PS: (request' = 2);
  [request3]   request = 0 -> RS/PS: (request' = 3);
  [request4]   request = 0 -> RS/PS: (request' = 4);
  [request5]   request = 0 -> RS/PS: (request' = 5);
  [response1] request = 1 -> RS/(1-PS): (request' = 0);
  [response2] request = 2 -> RS/(1-PS): (request' = 0);
  [response3] request = 3 -> RS/(1-PS): (request' = 0);
  [response4] request = 4 -> RS/(1-PS): (request' = 0);
  [response5] request = 5 -> RS/(1-PS): (request' = 0);
endmodule

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

// the number of clients waiting for acceptance of request
rewards "accept"
  client1 = 1 : 1 ;
  client2 = 1 : 1 ;
  client3 = 1 : 1 ;
  client4 = 1 : 1 ;
  client5 = 1 : 1 ;
endrewards

// the number of clients waiting for service of request
rewards "service"
```

```
  client1 = 2 : 1 ;
  client2 = 2 : 1 ;
  client3 = 2 : 1 ;
  client4 = 2 : 1 ;
  client5 = 2 : 1 ;
endrewards

// the number of clients waiting in total
rewards "waiting"
  client1 = 1 | client1 = 2 : 1 ;
  client2 = 1 | client2 = 2 : 1 ;
  client3 = 1 | client3 = 2 : 1 ;
  client4 = 1 | client4 = 2 : 1 ;
  client5 = 1 | client5 = 2 : 1 ;
endrewards

// the utilization of the server
rewards "utilization"
  request != 0 : 1;
endrewards

// the time spent in every state
rewards "time"
  true : 1 ;
endrewards


// -------------------------------------------------------------------
// ClientServer1.csl
// -------------------------------------------------------------------

// the expected number of clients in the respective states
"acceptNumber":  R{"accept"}=? [ S ];
"serviceNumber": R{"service"}=? [ S ];
"waitingNumber": R{"waiting"}=? [ S ];
"utilization":   R{"utilization"}=? [ S ];

// the expected times in the various states
"acceptTime": filter(avg, R{"time"}=? [ F client1=2 ], client1=1);

// the average time that a request is serviced
"serviceTime": filter(avg, R{"time"}=? [ F client1=0 ], client1=2);

// the average time that a client waits for a response
"waitingTime": filter(avg, R{"time"}=? [ F client1=0 ], client1=1);
```

C.2. Request Queueing

```
// -------------------------------------------------------------------
// ClientServer2.sm
// A system with multiple clients and one server where
// the client requests are queued.
//
// 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
// ----------------------------------------------------------------

const int N = 5; // number of clients

const double RC; // rate with which a client generates a request
const double RS; // rate with which the server handles a request
const double PS; // fraction of time server spends in acceptance

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

// the first client
module C1
  client1: [0..2] init 0;
  []          client1 = 0 -> RC : (client1' = 1);
  [request1]  client1 = 1 -> (client1' = 2);
  [response1] client1 = 2 -> (client1' = 0);
endmodule

// the other clients
module C2 = C1
  [ client1 = client2, request1 = request2, response1 = response2 ]
endmodule
module C3 = C1
  [ client1 = client3, request1 = request3, response1 = response3 ]
endmodule
module C4 = C1
  [ client1 = client4, request1 = request4, response1 = response4 ]
endmodule
module C5 = C1
  [ client1 = client5, request1 = request5, response1 = response5 ]
endmodule

// the queue (t denotes the position of the tail)
```

```
formula t = h+n < N ? h+n : h+n-N;
module Queue
  q0: [1..N] init 1;
  q1: [1..N] init 1;
  q2: [1..N] init 1;
  q3: [1..N] init 1;
  q4: [1..N] init 1;
  h: [0..N-1] init 0;
  n: [0..N]   init 0;

  [request1] n < 5 & t = 0 -> 2*RS/PS: (q0' = 1) & (n' = n+1);
  [request1] n < 5 & t = 1 -> 2*RS/PS: (q1' = 1) & (n' = n+1);
  [request1] n < 5 & t = 2 -> 2*RS/PS: (q2' = 1) & (n' = n+1);
  [request1] n < 5 & t = 3 -> 2*RS/PS: (q3' = 1) & (n' = n+1);
  [request1] n < 5 & t = 4 -> 2*RS/PS: (q4' = 1) & (n' = n+1);
  [forward1] n > 0 & h = 0 & q0 = 1 -> (h' = 1) & (n' = n-1);
  [forward1] n > 0 & h = 1 & q1 = 1 -> (h' = 2) & (n' = n-1);
  [forward1] n > 0 & h = 2 & q2 = 1 -> (h' = 3) & (n' = n-1);
  [forward1] n > 0 & h = 3 & q3 = 1 -> (h' = 4) & (n' = n-1);
  [forward1] n > 0 & h = 4 & q4 = 1 -> (h' = 0) & (n' = n-1);

  [request2] n < 5 & t = 0 -> 2*RS/PS: (q0' = 2) & (n' = n+1);
  [request2] n < 5 & t = 1 -> 2*RS/PS: (q1' = 2) & (n' = n+1);
  [request2] n < 5 & t = 2 -> 2*RS/PS: (q2' = 2) & (n' = n+1);
  [request2] n < 5 & t = 3 -> 2*RS/PS: (q3' = 2) & (n' = n+1);
  [request2] n < 5 & t = 4 -> 2*RS/PS: (q4' = 2) & (n' = n+1);
  [forward2] n > 0 & h = 0 & q0 = 2 -> (h' = 1) & (n' = n-1);
  [forward2] n > 0 & h = 1 & q1 = 2 -> (h' = 2) & (n' = n-1);
  [forward2] n > 0 & h = 2 & q2 = 2 -> (h' = 3) & (n' = n-1);
  [forward2] n > 0 & h = 3 & q3 = 2 -> (h' = 4) & (n' = n-1);
  [forward2] n > 0 & h = 4 & q4 = 2 -> (h' = 0) & (n' = n-1);

  [request3] n < 5 & t = 0 -> 2*RS/PS: (q0' = 3) & (n' = n+1);
  [request3] n < 5 & t = 1 -> 2*RS/PS: (q1' = 3) & (n' = n+1);
  [request3] n < 5 & t = 2 -> 2*RS/PS: (q2' = 3) & (n' = n+1);
  [request3] n < 5 & t = 3 -> 2*RS/PS: (q3' = 3) & (n' = n+1);
  [request3] n < 5 & t = 4 -> 2*RS/PS: (q4' = 3) & (n' = n+1);
  [forward3] n > 0 & h = 0 & q0 = 3 -> (h' = 1) & (n' = n-1);
  [forward3] n > 0 & h = 1 & q1 = 3 -> (h' = 2) & (n' = n-1);
  [forward3] n > 0 & h = 2 & q2 = 3 -> (h' = 3) & (n' = n-1);
  [forward3] n > 0 & h = 3 & q3 = 3 -> (h' = 4) & (n' = n-1);
  [forward3] n > 0 & h = 4 & q4 = 3 -> (h' = 0) & (n' = n-1);

  [request4] n < 5 & t = 0 -> 2*RS/PS: (q0' = 4) & (n' = n+1);
  [request4] n < 5 & t = 1 -> 2*RS/PS: (q1' = 4) & (n' = n+1);
  [request4] n < 5 & t = 2 -> 2*RS/PS: (q2' = 4) & (n' = n+1);
  [request4] n < 5 & t = 3 -> 2*RS/PS: (q3' = 4) & (n' = n+1);
  [request4] n < 5 & t = 4 -> 2*RS/PS: (q4' = 4) & (n' = n+1);
  [forward4] n > 0 & h = 0 & q0 = 4 -> (h' = 1) & (n' = n-1);
  [forward4] n > 0 & h = 1 & q1 = 4 -> (h' = 2) & (n' = n-1);
  [forward4] n > 0 & h = 2 & q2 = 4 -> (h' = 3) & (n' = n-1);
  [forward4] n > 0 & h = 3 & q3 = 4 -> (h' = 4) & (n' = n-1);
  [forward4] n > 0 & h = 4 & q4 = 4 -> (h' = 0) & (n' = n-1);

  [request5] n < 5 & t = 0 -> 2*RS/PS: (q0' = 5) & (n' = n+1);
  [request5] n < 5 & t = 1 -> 2*RS/PS: (q1' = 5) & (n' = n+1);
  [request5] n < 5 & t = 2 -> 2*RS/PS: (q2' = 5) & (n' = n+1);
  [request5] n < 5 & t = 3 -> 2*RS/PS: (q3' = 5) & (n' = n+1);
  [request5] n < 5 & t = 4 -> 2*RS/PS: (q4' = 5) & (n' = n+1);
  [forward5] n > 0 & h = 0 & q0 = 5 -> (h' = 1) & (n' = n-1);
  [forward5] n > 0 & h = 1 & q1 = 5 -> (h' = 2) & (n' = n-1);
  [forward5] n > 0 & h = 2 & q2 = 5 -> (h' = 3) & (n' = n-1);
  [forward5] n > 0 & h = 3 & q3 = 5 -> (h' = 4) & (n' = n-1);
  [forward5] n > 0 & h = 4 & q4 = 5 -> (h' = 0) & (n' = n-1);
endmodule

// the server
module Server
  request: [0..N] init 0;
  [forward1]  request = 0 -> 2*RS/PS: (request' = 1);
  [forward2]  request = 0 -> 2*RS/PS: (request' = 2);
  [forward3]  request = 0 -> 2*RS/PS: (request' = 3);
  [forward4]  request = 0 -> 2*RS/PS: (request' = 4);
  [forward5]  request = 0 -> 2*RS/PS: (request' = 5);
  [response1] request = 1 -> RS/(1-PS): (request' = 0);
  [response2] request = 2 -> RS/(1-PS): (request' = 0);
  [response3] request = 3 -> RS/(1-PS): (request' = 0);
  [response4] request = 4 -> RS/(1-PS): (request' = 0);
  [response5] request = 5 -> RS/(1-PS): (request' = 0);
endmodule

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

// the number of clients waiting for acceptance of request
rewards "accept"
  client1 = 1 : 1 ;
  client2 = 1 : 1 ;
  client3 = 1 : 1 ;
  client4 = 1 : 1 ;
  client5 = 1 : 1 ;
endrewards

// the number of clients waiting for service of request
rewards "service"
  client1 = 2 : 1 ;
  client2 = 2 : 1 ;
  client3 = 2 : 1 ;
  client4 = 2 : 1 ;
  client5 = 2 : 1 ;
endrewards

// the number of clients waiting in total
rewards "waiting"
  client1 = 1 | client1 = 2 : 1 ;
  client2 = 1 | client2 = 2 : 1 ;
  client3 = 1 | client3 = 2 : 1 ;
  client4 = 1 | client4 = 2 : 1 ;
  client5 = 1 | client5 = 2 : 1 ;
endrewards

// the utilization of the server
rewards "utilization"
  request != 0 : 1;
endrewards

// the time spent in every state
rewards "time"
  true : 1 ;
endrewards


// ----------------------------------------------------------------------
// ClientServer2.csl
// ----------------------------------------------------------------------

// the expected number of clients in the respective states
"acceptNumber":  R{"accept"}=? [ S ];
"serviceNumber": R{"service"}=? [ S ];
"waitingNumber": R{"waiting"}=? [ S ];
"utilization":   R{"utilization"}=? [ S ];

// the expected times in the various states
"acceptTime": filter(avg, R{"time"}=? [ F client1=2 ], client1=1);

// the average time that a request is serviced
"serviceTime": filter(avg, R{"time"}=? [ F client1=0 ], client1=2);

// the average time that a client waits for a response
"waitingTime": filter(avg, R{"time"}=? [ F client1=0 ], client1=1);
"queueTime":   "waitingTime"-(1-PS)/RS;
```

C.3. The Queue-Based Model Revisited

```
// ---------------------------------------------------------------------
// ClientServer3.sm
// A system with multiple clients and one server where
// the client requests are queued and clients are not
// synchronized with responses.
//
// 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
// -------------------------------------------------------------------

const int N = 5; // number of clients

const double RC; // rate with which a client generates a request
const double RS; // rate with which the server handles a request
const double PS; // fraction of time server spends in acceptance

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

// the first client
module C1
  client1: [0..1] init 0;
  []          client1 = 0 -> RC : (client1' = 1);
  [request1]  client1 = 1 -> (client1' = 0);
endmodule

// the other clients
module C2 = C1
  [ client1 = client2, request1 = request2 ]
endmodule
module C3 = C1
  [ client1 = client3, request1 = request3 ]
endmodule
module C4 = C1
  [ client1 = client4, request1 = request4 ]
endmodule
module C5 = C1
  [ client1 = client5, request1 = request5 ]
endmodule

// the queue (t denotes the position of the tail)
formula t = h+n < N ? h+n : h+n-N;
```

```
module Queue
  q0: [1..N] init 1;
  q1: [1..N] init 1;
  q2: [1..N] init 1;
  q3: [1..N] init 1;
  q4: [1..N] init 1;
  h: [0..N-1] init 0;
  n: [0..N]   init 0;
  accepted: bool init true;

  [request1] n = 5 -> 2*RS/PS: (accepted' = false);
  [request1] n < 5 & t = 0 -> 2*RS/PS:
    (q0' = 1) & (n' = n+1) & (accepted' = true);
  [request1] n < 5 & t = 1 -> 2*RS/PS:
    (q1' = 1) & (n' = n+1) & (accepted' = true);
  [request1] n < 5 & t = 2 -> 2*RS/PS:
     (q2' = 1) & (n' = n+1) & (accepted' = true);
  [request1] n < 5 & t = 3 -> 2*RS/PS:
     (q3' = 1) & (n' = n+1) & (accepted' = true);
  [request1] n < 5 & t = 4 -> 2*RS/PS:
     (q4' = 1) & (n' = n+1) & (accepted' = true);
  [forward1] n > 0 & h = 0 & q0 = 1 -> (h' = 1) & (n' = n-1);
  [forward1] n > 0 & h = 1 & q1 = 1 -> (h' = 2) & (n' = n-1);
  [forward1] n > 0 & h = 2 & q2 = 1 -> (h' = 3) & (n' = n-1);
  [forward1] n > 0 & h = 3 & q3 = 1 -> (h' = 4) & (n' = n-1);
  [forward1] n > 0 & h = 4 & q4 = 1 -> (h' = 0) & (n' = n-1);

  [request2] n = 5 -> 2*RS/PS: (accepted' = false);
  [request2] n < 5 & t = 0 -> 2*RS/PS:
    (q0' = 2) & (n' = n+1) & (accepted' = true);
  [request2] n < 5 & t = 1 -> 2*RS/PS:
    (q1' = 2) & (n' = n+1) & (accepted' = true);
  [request2] n < 5 & t = 2 -> 2*RS/PS:
    (q2' = 2) & (n' = n+1) & (accepted' = true);
  [request2] n < 5 & t = 3 -> 2*RS/PS:
    (q3' = 2) & (n' = n+1) & (accepted' = true);
  [request2] n < 5 & t = 4 -> 2*RS/PS:
    (q4' = 2) & (n' = n+1) & (accepted' = true);
  [forward2] n > 0 & h = 0 & q0 = 2 -> (h' = 1) & (n' = n-1);
  [forward2] n > 0 & h = 1 & q1 = 2 -> (h' = 2) & (n' = n-1);
  [forward2] n > 0 & h = 2 & q2 = 2 -> (h' = 3) & (n' = n-1);
  [forward2] n > 0 & h = 3 & q3 = 2 -> (h' = 4) & (n' = n-1);
  [forward2] n > 0 & h = 4 & q4 = 2 -> (h' = 0) & (n' = n-1);

  [request3] n = 5 -> 2*RS/PS: (accepted' = false);
  [request3] n < 5 & t = 0 -> 2*RS/PS:
    (q0' = 3) & (n' = n+1) & (accepted' = true);
  [request3] n < 5 & t = 1 -> 2*RS/PS:
    (q1' = 3) & (n' = n+1) & (accepted' = true);
  [request3] n < 5 & t = 2 -> 2*RS/PS:
    (q2' = 3) & (n' = n+1) & (accepted' = true);
  [request3] n < 5 & t = 3 -> 2*RS/PS:
    (q3' = 3) & (n' = n+1) & (accepted' = true);
  [request3] n < 5 & t = 4 -> 2*RS/PS:
    (q4' = 3) & (n' = n+1) & (accepted' = true);
  [forward3] n > 0 & h = 0 & q0 = 3 -> (h' = 1) & (n' = n-1);
  [forward3] n > 0 & h = 1 & q1 = 3 -> (h' = 2) & (n' = n-1);
  [forward3] n > 0 & h = 2 & q2 = 3 -> (h' = 3) & (n' = n-1);
  [forward3] n > 0 & h = 3 & q3 = 3 -> (h' = 4) & (n' = n-1);
  [forward3] n > 0 & h = 4 & q4 = 3 -> (h' = 0) & (n' = n-1);

  [request4] n = 5 -> 2*RS/PS: (accepted' = false);
  [request4] n < 5 & t = 0 -> 2*RS/PS:
    (q0' = 4) & (n' = n+1) & (accepted' = true);
  [request4] n < 5 & t = 1 -> 2*RS/PS:
    (q1' = 4) & (n' = n+1) & (accepted' = true);
  [request4] n < 5 & t = 2 -> 2*RS/PS:
    (q2' = 4) & (n' = n+1) & (accepted' = true);
  [request4] n < 5 & t = 3 -> 2*RS/PS:
    (q3' = 4) & (n' = n+1) & (accepted' = true);
  [request4] n < 5 & t = 4 -> 2*RS/PS:
    (q4' = 4) & (n' = n+1) & (accepted' = true);
  [forward4] n > 0 & h = 0 & q0 = 4 -> (h' = 1) & (n' = n-1);
  [forward4] n > 0 & h = 1 & q1 = 4 -> (h' = 2) & (n' = n-1);
  [forward4] n > 0 & h = 2 & q2 = 4 -> (h' = 3) & (n' = n-1);
  [forward4] n > 0 & h = 3 & q3 = 4 -> (h' = 4) & (n' = n-1);
  [forward4] n > 0 & h = 4 & q4 = 4 -> (h' = 0) & (n' = n-1);

  [request5] n = 5 -> 2*RS/PS: (accepted' = false);
  [request5] n < 5 & t = 0 -> 2*RS/PS:
    (q0' = 5) & (n' = n+1) & (accepted' = true);
  [request5] n < 5 & t = 1 -> 2*RS/PS:
    (q1' = 5) & (n' = n+1) & (accepted' = true);
  [request5] n < 5 & t = 2 -> 2*RS/PS:
    (q2' = 5) & (n' = n+1) & (accepted' = true);
  [request5] n < 5 & t = 3 -> 2*RS/PS:
    (q3' = 5) & (n' = n+1) & (accepted' = true);
  [request5] n < 5 & t = 4 -> 2*RS/PS:
    (q4' = 5) & (n' = n+1) & (accepted' = true);
  [forward5] n > 0 & h = 0 & q0 = 5 -> (h' = 1) & (n' = n-1);
  [forward5] n > 0 & h = 1 & q1 = 5 -> (h' = 2) & (n' = n-1);
  [forward5] n > 0 & h = 2 & q2 = 5 -> (h' = 3) & (n' = n-1);
  [forward5] n > 0 & h = 3 & q3 = 5 -> (h' = 4) & (n' = n-1);
  [forward5] n > 0 & h = 4 & q4 = 5 -> (h' = 0) & (n' = n-1);
endmodule

// the server
module Server
```

```
  request: [0..N] init 0;
  [forward1] request = 0 -> 2*RS/PS: (request' = 1);
  [forward2] request = 0 -> 2*RS/PS: (request' = 2);
  [forward3] request = 0 -> 2*RS/PS: (request' = 3);
  [forward4] request = 0 -> 2*RS/PS: (request' = 4);
  [forward5] request = 0 -> 2*RS/PS: (request' = 5);
  [] request != 0 -> RS/(1-PS): (request' = 0);
endmodule

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

// the number of clients waiting for acceptance of request
rewards "accept"
  client1 = 1 : 1 ;
  client2 = 1 : 1 ;
  client3 = 1 : 1 ;
  client4 = 1 : 1 ;
  client5 = 1 : 1 ;
endrewards

// the utilization of the queue
rewards "waiting"
  true : n;
endrewards

// the utilization of the server
rewards "utilization"
  request != 0 : 1;
endrewards

// the probability of a request getting rejected
rewards "rejected"
  !accepted : 1;
endrewards

// the time spent in every state
rewards "time"
  true : 1 ;
endrewards


// -------------------------------------------------------------------
// ClientServer3.csl
// -------------------------------------------------------------------

// the expected number of clients in the respective states
"accept": R{"accept"}=? [ S ];
```

```
// the utilization of the server
"utilization": R{"utilization"}=? [ S ];

// the utilization of the queue
"waiting": R{"waiting"}=? [ S ];

// the rejection probability
"rejected": R{"rejected"}=? [ S ];

// the average time till a request is serviced
"acceptTime0": filter(avg, R{"time"}=? [ F request=1 ], client1=1);
"acceptTime1":  "acceptTime0"/(1-"rejected");

// the average time till a request is fulfilled
"waitingTime0": "acceptTime0"+(1-PS)/RS;
"waitingTime1": "acceptTime1"+(1-PS)/RS;
```

C.4. The Queue-Based Model Revisited Again

```
// -------------------------------------------------------------------
// ClientServer6.sm
// A system with multiple clients and one server where
// the client requests are queued and clients are not
// synchronized with responses.
//
// 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
// -------------------------------------------------------------------

const int N = 5; // number of clients

const double RC; // rate with which a client generates a request
const double RS; // rate with which the server handles a request

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

// the first client
module C1
  [request1] true -> RC : true;
  [reject1]  true -> RC : true;
endmodule

// the other clients
module C2 = C1
  [ request1 = request2, reject1 = reject2 ]
endmodule
module C3 = C1
  [ request1 = request3, reject1 = reject3 ]
endmodule
module C4 = C1
  [ request1 = request4, reject1 = reject4 ]
endmodule
module C5 = C1
  [ request1 = request5, reject1 = reject5 ]
endmodule

// the queue (t denotes the position of the tail)
formula t = h+n < N ? h+n : h+n-N;
module Queue
  q0: [1..N] init 1;
  q1: [1..N] init 1;
  q2: [1..N] init 1;
  q3: [1..N] init 1;
  q4: [1..N] init 1;
  h: [0..N-1] init 0;
  n: [0..N]   init 0;
  r: [0..N]   init 0;

  [reject1] n = 5 -> true;
  [request1] n < 5 & t = 0 -> (q0' = 1) & (n' = n+1) & (r' = 1);
  [request1] n < 5 & t = 1 -> (q1' = 1) & (n' = n+1) & (r' = 1);
  [request1] n < 5 & t = 2 -> (q2' = 1) & (n' = n+1) & (r' = 1);
  [request1] n < 5 & t = 3 -> (q3' = 1) & (n' = n+1) & (r' = 1);
  [request1] n < 5 & t = 4 -> (q4' = 1) & (n' = n+1) & (r' = 1);
  [forward1] n > 0 & h = 0 & q0 = 1 -> (h' = 1) & (n' = n-1);
  [forward1] n > 0 & h = 1 & q1 = 1 -> (h' = 2) & (n' = n-1);
  [forward1] n > 0 & h = 2 & q2 = 1 -> (h' = 3) & (n' = n-1);
  [forward1] n > 0 & h = 3 & q3 = 1 -> (h' = 4) & (n' = n-1);
  [forward1] n > 0 & h = 4 & q4 = 1 -> (h' = 0) & (n' = n-1);

  [reject2] n = 5 -> true;
  [request2] n < 5 & t = 0 -> (q0' = 2) & (n' = n+1) & (r' = 2);
  [request2] n < 5 & t = 1 -> (q1' = 2) & (n' = n+1) & (r' = 2);
  [request2] n < 5 & t = 2 -> (q2' = 2) & (n' = n+1) & (r' = 2);
  [request2] n < 5 & t = 3 -> (q3' = 2) & (n' = n+1) & (r' = 2);
  [request2] n < 5 & t = 4 -> (q4' = 2) & (n' = n+1) & (r' = 2);
  [forward2] n > 0 & h = 0 & q0 = 2 -> (h' = 1) & (n' = n-1);
  [forward2] n > 0 & h = 1 & q1 = 2 -> (h' = 2) & (n' = n-1);
  [forward2] n > 0 & h = 2 & q2 = 2 -> (h' = 3) & (n' = n-1);
  [forward2] n > 0 & h = 3 & q3 = 2 -> (h' = 4) & (n' = n-1);
  [forward2] n > 0 & h = 4 & q4 = 2 -> (h' = 0) & (n' = n-1);

  [reject3] n = 5 -> true;
  [request3] n < 5 & t = 0 -> (q0' = 3) & (n' = n+1) & (r' = 3);
  [request3] n < 5 & t = 1 -> (q1' = 3) & (n' = n+1) & (r' = 3);
  [request3] n < 5 & t = 2 -> (q2' = 3) & (n' = n+1) & (r' = 3);
  [request3] n < 5 & t = 3 -> (q3' = 3) & (n' = n+1) & (r' = 3);
  [request3] n < 5 & t = 4 -> (q4' = 3) & (n' = n+1) & (r' = 3);
  [forward3] n > 0 & h = 0 & q0 = 3 -> (h' = 1) & (n' = n-1);
  [forward3] n > 0 & h = 1 & q1 = 3 -> (h' = 2) & (n' = n-1);
  [forward3] n > 0 & h = 2 & q2 = 3 -> (h' = 3) & (n' = n-1);
  [forward3] n > 0 & h = 3 & q3 = 3 -> (h' = 4) & (n' = n-1);
  [forward3] n > 0 & h = 4 & q4 = 3 -> (h' = 0) & (n' = n-1);

  [reject4] n = 5 -> true;
  [request4] n < 5 & t = 0 -> (q0' = 4) & (n' = n+1) & (r' = 4);
  [request4] n < 5 & t = 1 -> (q1' = 4) & (n' = n+1) & (r' = 4);
  [request4] n < 5 & t = 2 -> (q2' = 4) & (n' = n+1) & (r' = 4);
  [request4] n < 5 & t = 3 -> (q3' = 4) & (n' = n+1) & (r' = 4);
  [request4] n < 5 & t = 4 -> (q4' = 4) & (n' = n+1) & (r' = 4);
  [forward4] n > 0 & h = 0 & q0 = 4 -> (h' = 1) & (n' = n-1);
  [forward4] n > 0 & h = 1 & q1 = 4 -> (h' = 2) & (n' = n-1);
  [forward4] n > 0 & h = 2 & q2 = 4 -> (h' = 3) & (n' = n-1);
  [forward4] n > 0 & h = 3 & q3 = 4 -> (h' = 4) & (n' = n-1);
  [forward4] n > 0 & h = 4 & q4 = 4 -> (h' = 0) & (n' = n-1);

  [reject5] n = 5 -> true;
  [request5] n < 5 & t = 0 -> (q0' = 5) & (n' = n+1) & (r' = 5);
  [request5] n < 5 & t = 1 -> (q1' = 5) & (n' = n+1) & (r' = 5);
  [request5] n < 5 & t = 2 -> (q2' = 5) & (n' = n+1) & (r' = 5);
  [request5] n < 5 & t = 3 -> (q3' = 5) & (n' = n+1) & (r' = 5);
  [request5] n < 5 & t = 4 -> (q4' = 5) & (n' = n+1) & (r' = 5);
  [forward5] n > 0 & h = 0 & q0 = 5 -> (h' = 1) & (n' = n-1);
  [forward5] n > 0 & h = 1 & q1 = 5 -> (h' = 2) & (n' = n-1);
  [forward5] n > 0 & h = 2 & q2 = 5 -> (h' = 3) & (n' = n-1);
  [forward5] n > 0 & h = 3 & q3 = 5 -> (h' = 4) & (n' = n-1);
  [forward5] n > 0 & h = 4 & q4 = 5 -> (h' = 0) & (n' = n-1);
endmodule

// the server
module Server
  request: [0..N] init 0;
  [forward1] true -> RS: (request' = 1);
  [forward2] true -> RS: (request' = 2);
  [forward3] true -> RS: (request' = 3);
  [forward4] true -> RS: (request' = 4);
  [forward5] true -> RS: (request' = 5);
```

```
endmodule

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

// the number of elements in the queue
rewards "waiting"
  true : n;
endrewards

// the number of rejection events
rewards "rejected"
  [reject1] true : 1;
  [reject2] true : 1;
  [reject3] true : 1;
  [reject4] true : 1;
  [reject5] true : 1;
endrewards

// the number of acceptance events
rewards "accepted"
  [request1] true : 1;
  [request2] true : 1;
  [request3] true : 1;
  [request4] true : 1;
  [request5] true : 1;
endrewards

// the time spent in every state
rewards "time"
  true : 1 ;
endrewards


// ----------------------------------------------------------------
// ClientServer6.csl
// ----------------------------------------------------------------

// the number of elements in the queue
"waiting":  R{"waiting"}=? [ S ];

// the rejection probability
"rejectedN": R{"rejected"}=? [ S ];
"acceptedN": R{"accepted"}=? [ S ];
"rejected":  1/(1+"acceptedN"/"rejectedN");

// the average time till a request is serviced
"acceptTime": filter(avg, R{"time"}=? [ F request=1 ], r=1);
```

```
// the average time till a request is fulfilled
"waitingTime": "acceptTime"+1/RS;
```