Experiments with Measuring Time in PRISM 4.0 (Addendum)*

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

April 5, 2013

Abstract

In our previous paper "Experiments with Measuring Time in PRISM 4.0" we have reported on experiments with the probabilistic symbol model checker PRISM 4.0 on evaluating the response times of client/server system models. In that report some questions remained unresolved, in particular an unexpected result in the analysis of Probabilistic Timed Automata (PTA) in PRISM, and an unexplained discrepancy between explicit time measurement and results derived from queueing theory (by application of Little's Law) in the analysis of a simple client/server model. In this addendum to that paper these open questions are addressed and (essentially) solved. In particular, it is shown that expected time analysis by explicit time measurement in PRISM is more complex than previously thought such that the application of Little's Law still has its merit.

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

Contents

1. Introduction

This paper represents an addendum to our previous report [1] whose content is summarized by the respective 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.

However, that paper left two questions open:

1. In Section 3.4 "Discrete Time Models" we reported on an anomaly in the evaluation of Probabilistic Timed Automata which we did not understand.
2. In Section 4.4 "The Queue-Based Model Revisited Again" we reported on a small discrepancy between the average time measured between putting a message in the queue and removing it from the queue and the time expected by the application of "Little's Law".

These questions are now going to be addressed and (essentially) resolved. In Section 2, we give the simple explanation of the first anomaly, while Section 3 gives the more subtle explanation and a solution to the second problem. In Section 4 we revise our conclusions of the previous report based on the new results. Appendix A lists the new PRISM model and associated queries introduced in this addendum.

The author thanks David Parker and Janos Sztrik for helping in short discussions to solve these questions.

2. Probabilistic Timed Automata

The problem raised in Section 3.4 "Discrete Time Models" of [1] was that for the Probabilistic Timed Automata (PTA) Model

```
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 $[T_1, T_2]$ and the reward structure

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

which associates reward 1 to every state, the query

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

reported for $T_1 = 80$ and $T_2 = 120$ the results 200 and 300 rather than the expected values 80 and 120. The model was subsequently reduced to a simpler one for which corresponding queries with different values of T_1 and T_2 produced strange results.

We have contacted David Parker, leader of the development of PRISM, who has confirmed that the results were the consequence of a bug in PRISM 4.0. With the newer PRISM development version 4.1.beta.r6275 available since December 2012 from the PRISM web site the expected result 80 and 120 could be produced.

3. The Queue-Based Model Revisited Once Again

The problem raised in Section 4.4 "The Queue-Based Model Revisited Again" of [1] arose in the context of the following queue-based client-server model:

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

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

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
```

Here $N = 5$ clients submit with rate $RC = 10$ requests into a queue of size N from where they are served by a server with rate $RS \in [60, 100]$. For this model, we measured with the reward

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

and the query

```
waiting": R{"waiting"}=? [ S ] / (N*RC);
```

the quotient of the average number of requests that are waiting in the queue and of the rate with which requests are submitted to the queue; applying "Little's Law" the result is the expected time that a request spends in the queue; this measured result also conformed exactly to the result of the analytical analysis performed in Section 1 of [1].

Using the reward

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

we expected to be able to derive the same time by the query

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

which measures the average time between a state satisfying $r = 1$ (a request of client 1 is submitted to the queue) and a state satisfying $request = 1$ (a request from client 1 has been processed by the server). However, while the results were close to the ones presented above, they were not identical (the query result slightly underestimated the time for small *RS* and slightly overestimated it for large *RS*). While the differences were not overwhelming, they were still significant and indicated a discrepancy between the model above and the classical queue model which we were not able to explain at the time of writing [1].

However, in the course of a discussion with Janos Sztrik, the author realized that the query above indeed does not accurately measure the average time that a request from client 1 spends in the queue: since the client does not wait for the server to have the query processed before submitting a request, it is possible that multiple requests from client 1 are in the queue, and thus the query also considers the (shorter) times from the submission of one query to the processing of a previously submitted query (there is another more fundamental reason why above query is inadequate, which will be explained below).

The problem is thus how to identify a particular query uniquely, if the client identifier stored in the request cannot serve that purpose. Rethinking the problem, the author realized that the position of the query in the queue can serve as a unique identifier and that the content of the request itself is not needed for measuring the execution time. Since in our simple model also the server does not require the content, we drop it altogether, which yields a much simpler model with a small state space which can be quickly analyzed by PRISM (see Appendix A for the full source):

```
module Client
  [request] true -> RC : true;
  [reject]  true -> RC : true;
endmodule

formula t = h+n < N ? h+n : h+n-N;
module Queue
  h: [0..N-1] init 0;
  n: [0..N]   init 0;
  [reject]  n = N -> true;
  [request] n < N -> (n' = n+1);
  [forward] n > 0 -> (h' = h+1 < N ? h+1 : 0) & (n' = n-1);
endmodule

module Server
  [forward] true -> RS : true;
endmodule
```

Since the messages do not carry values any more, we just use a single client component that represents the collection of all clients; the request submission rate has to be correspondingly adjusted (rather than 5 components with submission rate $RC = 10$, we have now one component with submission rate $RC = 50$).

The queue component now just models the position h of the head (from which the next request is delivered to the server) and the number n of requests in the queue; the position t of the tail (where the next request from the client is to be stored) can be derived from h and n by simple calculation. As in the previous model, to save space, we also do not explicitly store the status of the last request submitted from the client (accepted or rejected) but indicate this by different transitions "request" and "reject". For this model, with the reward structures

```
rewards "rejected"
  [reject] true : 1;
endrewards

rewards "accepted"
  [request] true : 1;
endrewards
```

and the queries

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

we can calculate the steady state expectation values of the number of accepted and rejected requests and, e.g., compute from this the rejection probability of a request (we have verified that the same probability is derived as by using an explicit state variable).

Furthermore, using the reward structure

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

which assigns to every state as a reward the number of requests in the queue, the queries

```
"time0": "waiting"/RC;
"time1": "time0"*(1+"rejectedN"/"acceptedN");
```

apply Little's Law to determine the average time "time0" that a request spends in the queue and the average time "time1" that a request in total waits to being served (considering the probability of being rejected by the queue). The results conform exactly to the analytical evaluation performed in Section 2 of [1].

However, since our goal is to measure this time directly, we apply the usual time reward structure

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

and perform the queries

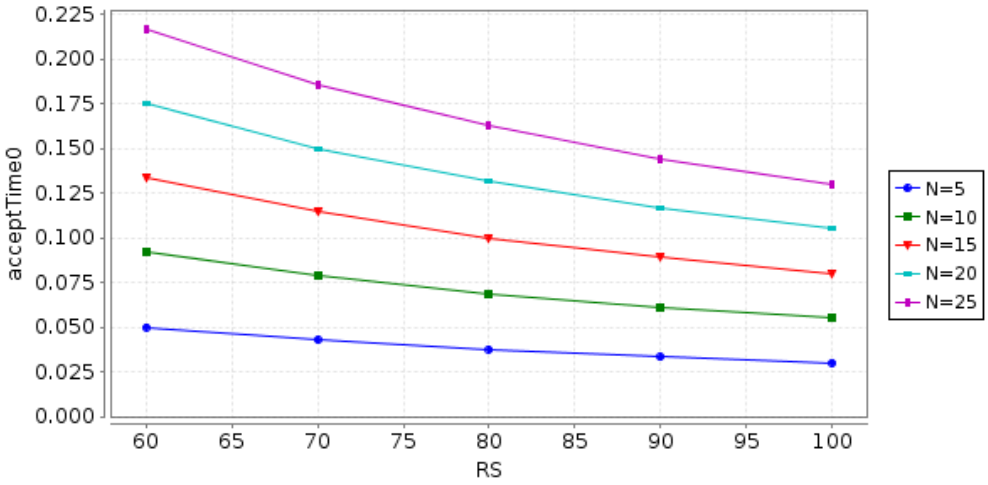

Figure 1: Analyzing the Acceptance Time (Wrong Attempt)

```
"acceptTime0":
  filter(avg, R{"time"}=? [ F (h=V & n<N) ], t=V & n>0);
"acceptTime1":
  "acceptTime0"*(1+"rejectedN"/"acceptedN");
```

Here "acceptTime0" measures the average time between a state in which the tail t has some value V and the queue is not empty and a state in which the head h has value V and the queue is not full: the first state arises immediately after inserting a new request into the queue and advancing t to V; the second state arises immediately after this request has been removed and h is advanced to V (the side conditions $n > 0$ and $n < N$ are needed to distinguish these states from those where $h = t$ holds because a queue is empty respectively full). Query "acceptTime1" adjusts as usual the time spent in the queue to the total time for a request to get served.

As expected the results of these queries are independent of the choice of $V \in [0, N-1]$. However, they depend on the choice of N as depicted in Figure 1. Not only that the results for $N = 5$ overestimate the values derived from the application of Little's Law, we see that for growing values of queue size N, the acceptance times do not converge, as would be expected from the theoretical analysis (see Section 2 of [1]). Something in our query is fundamentally wrong.

The crucial hint towards the error is that the result of the query grows linearly with the queue size. Thus apparently the worst-case of a query being inserted into a queue with $N-1$ elements contributes significantly to the average acceptance time, in contrast to our understanding that such cases become for growing N more and more unlikely and should not contribute much to the time any more. This is the point where we realize our error: the query

```
"acceptTime0":
  filter(avg, R{"time"}=? [ F (h=V & n<N) ], t=V & n>0);
```

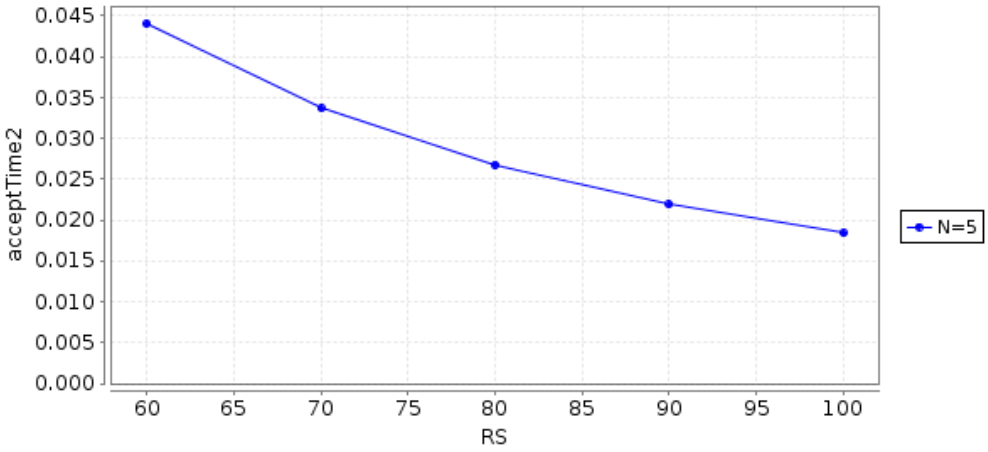

Figure 2: Analyzing the Waiting Time

computes the arithmetic mean of the rewards accumulated along all execution paths from states satisfying $t = V \wedge n > 0$, i.e., $t = V \wedge (n = 1 \vee n = 2 \vee \ldots \vee n = N)$ without considering that the probability of a state satisfying $n = i$ becomes lower for growing i. So the problem is that we have to consider the different probabilities of the different states, i.e., we have to construct classes of states with same probabilities, analyze these classes separately, and then compute the *weighted mean* of the results with respect to their relative weights.

In more detail, if P is the probability of being in a state satisfying $t = V \wedge n > 1$, p_i is the probability of being in a state with $t = V \wedge n = i$, and r_i is the expected reward for such a state, we have to calculate the expected reward R as

$$R = \sum_{i=1}^{N} \frac{p_i}{P} \cdot r_i = \frac{1}{P} \cdot \sum_{i=1}^{N} p_i \cdot r_i$$

where the relative weight p_i/P attributed to r_i is the conditional probability of being in state with $t = V \wedge n = i$, if being in a state with $t = V \wedge n > 1$; we then have $\sum_{i=1}^{N}(p_i/P) = 1$.

We can calculate the value of R for $N = 5$ in PRISM by the query

```
"acceptTime2": (
  filter(avg, R{"time"}=? [ F (h=V & n<N) ], t=V & n=1)
  * S=? [ t=V & n=1 ]
+ filter(avg, R{"time"}=? [ F (h=V & n<N) ], t=V & n=2)
  * S=? [ t=V & n=2 ]
+ filter(avg, R{"time"}=? [ F (h=V & n<N) ], t=V & n=3)
  * S=? [ t=V & n=3 ]
+ filter(avg, R{"time"}=? [ F (h=V & n<N) ], t=V & n=4)
  * S=? [ t=V & n=4 ]
+ filter(avg, R{"time"}=? [ F (h=V & n<N) ], t=V & n=5)
  * S=? [ t=V & n=5 ]
) / S=? [ t=V & n>0 ];
```

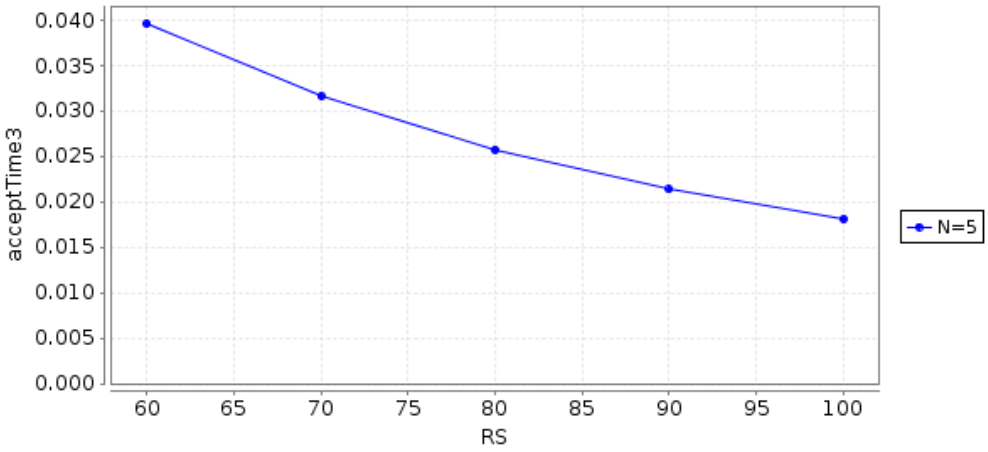

Figure 3: Analyzing the Acceptance Time

which gives the results depicted in Figure 2. These results are identical to those derived above from the application of Little's Law (up to 6 digits), however for the query "time1" which denotes the total waiting time (including failed attempts to enter the queue), not for the query "time0" which denotes the time actually spent in the queue. Only by multiplying with P_a where $P_a = a/(a+r) = 1/(1+r/a)$ is the probability that a request is accepted and a and r denote the number of requests that were accepted respectively rejected, i.e., by performing the query

```
"acceptTime3": "acceptTime2" / (1+"rejectedN"/"acceptedN");
```

we get the same result as "time0", see Figure 3.

Apparently we should have used in our calculation of R the base probability P/P_a, i.e., the conditional probability of a state satisfying $t = V \wedge n > 1$, if a message is accepted, but we admittedly lack any explanation why this is necessary. Thus while we can accurately reproduce the results of the analytical time analysis by explicit time measurement, there still remains a gap in our explanation to be addressed.

4. Conclusions

By this addendum to [1], we could solve most of the questions left open in that paper, first by showing that a problem with the analysis of probabilistic timed automata was due to a bug in PRISM 4.0 (resolved in the development version of the upcoming PRISM 4.1) and second by accurately reproducing the result of the expected time analysis of a queue model via explicit time measurement (still there remains an open question to be addressed with respect to the theoretical explanation of this result).

Equally important, however, are the following observations: first, we realized that queue models can be modeled in three levels of accuracy (rather than the previously used two levels):

1. By only a counter n that models the number of requests waiting in the queue. This is sufficient if only Little's Law is to be applied for expected time analysis and yields the model with the smallest number of states.

2. By the counter n and an index h that models the position of the head of the queue (in a virtual array representing the queue); from both n and h, also the virtual index t of the position of the tail of the queue can be derived. This is sufficient, if we want to determine the time a particular message spends in the queue, i.e., to use explicit time measurement rather than Little's Law; the model is still very small and can be quickly analyzed.

3. By the counter n, the index h, and variables $q_1, \ldots, q_N$ that model the actual contents of the queue (as a substitution for an array which is not directly supported by PRISM). This is necessary, if the behavior of the server depends on the actual value of the request, e.g., by returning a result to the client that submitted the request. The model thus becomes very large which can make actual model checking with PRISM ineffective.

Furthermore, we realized that expected time analysis by explicit time measurement is more complex than assumed in [1]. It is true that minimum and maximum times can be nicely computed by queries of form

```
filter(op, R{"time"}=? [ F target ], source);
```

where $op \in \{\texttt{min},\texttt{max}\}$, *source* is the property describing the initial state of a path, *target* is the property describing the final state of the path, and *time* is a reward structure used for calculating the time spent along that path.

However, for expected time analysis, we cannot indiscriminately choose $op = \texttt{avg}$ as it was done in Section 4 of [1]; this is only possible, if all states described by *source* have equal probability. If this is not the case, the set of states has to be split into subsets of states with same probability; for each of these subsets an individual query has to be performed. The expected time can be then derived by a weighted mean of the individual query results using the conditional probabilities of the individual classes as weights. The naive average time analysis performed in Sections 4.2–4.4 of [1] does thus not produce the expected results and must be revised along the lines sketched in this addendum.

However, our new approach of using weighted means makes expected time analysis by explicit time measurement in PRISM less attractive than previously thought (while best and worst case analysis are still easily possible by direct measurement, expected time analysis can become quite complex and error-prone); thus an indirect derivation by Little's Law seems still of merit for more complex models than the simple one analyzed in this addendum.

We will investigate these questions further with attempts to the analysis of more complex performance models.

References

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

A. The Queue-Based Model Revisited Once Again

```
// ------------------------------------------------------------------
// ClientServer8.sm
// A system with one client and one 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
// ------------------------------------------------------------------

const int N;     // size of queue
const double RC; // rate with which the client generates a request
const double RS; // rate with which the server handles a request

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

// the client
module Client
  [request] true -> RC : true;
  [reject]  true -> RC : true;
endmodule

// the queue (t denotes the position of the tail)
formula t = h+n < N ? h+n : h+n-N;
module Queue
  h: [0..N-1] init 0;
  n: [0..N]   init 0;
  [reject]  n = N -> true;
  [request] n < N -> (n' = n+1);
  [forward] n > 0 -> (h' = h+1 < N ? h+1 : 0) & (n' = n-1);
```

```
endmodule

// the server
module Server
  [forward] true -> RS : true;
endmodule

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

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

// the number of rejection events
rewards "rejected"
  [reject] true : 1;
endrewards

// the number of acceptance events
rewards "accepted"
  [request] true : 1;
endrewards

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

```
// ------------------------------------------------------------------
// ClientServer8.csl
// ------------------------------------------------------------------

const int V;

// 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 (plain average)
"acceptTime0": filter(avg, R{"time"}=? [ F (h=V & n<N) ], t=V & n>0);
"acceptTime1":  "acceptTime0"*(1+"rejectedN"/"acceptedN");

// the average time till a request is service (weighted average)
```

```
"acceptTime2": (
  filter(avg, R{"time"}=? [ F (h=V & n<N) ], t=V & n=1)
  * S=? [ t=V & n=1 ]
+ filter(avg, R{"time"}=? [ F (h=V & n<N) ], t=V & n=2)
  * S=? [ t=V & n=2 ]
+ filter(avg, R{"time"}=? [ F (h=V & n<N) ], t=V & n=3)
  * S=? [ t=V & n=3 ]
+ filter(avg, R{"time"}=? [ F (h=V & n<N) ], t=V & n=4)
  * S=? [ t=V & n=4 ]
+ filter(avg, R{"time"}=? [ F (h=V & n<N) ], t=V & n=5)
  * S=? [ t=V & n=5 ]
) / S=? [ t=V & n > 0 ];
"acceptTime3": "acceptTime2" / (1+"rejectedN"/"acceptedN");

// the same time computed according to Little's Law
"time0": "waiting"/RC;
"time1": "time0"*(1+"rejectedN"/"acceptedN");
```