Publ. Math. Debrecen
66/1-2 (2005), 197–211

Tool supported performance modelling of finite-source retrial queues with breakdowns

By JÁNOS SZTRIK (Debrecen)

To the memory of Professor Béla Brindza

Abstract. In this paper the tool supported performance modelling of a single server homogeneous finite-source retrial queueing system is presented. The server is assumed to be subject to random breakdowns depending on whether it is busy or idle. The failure of the server may block or unblock the system's operations and the service of the interrupted request may be resumed or the call can be transmitted to the orbit. All random variables involved in the model constructions are supposed to be exponentially distributed and independent of each other.

The novelty of investigations is the different type of non-reliability of the server. The MOSEL (Modeling, Specification and Evaluation Language) tool, developed at the University of Erlangen, Germany, was used to formulate and solve the problem and the main performance and reliability measures were derived and graphically displayed. Several numerical calculations were performed to show the effect of the non-reliability of the server on the mean response times of the calls, the overall utilization of the system, and the mean number of calls staying at the server or in the orbit.

Mathematics Subject Classification: 60K25, 68M20, 60J27.
Key words and phrases: retrial queueing systems, finite number of sources, non-reliable server, performance tool, performance and reliability measures, tool support.
Research is partially supported by Hungarian Scientific Research Fund OTKA 34280/2000.

1. Introduction

Over the last decade considerable effort has been put in the development of techniques to assess the performance and the dependability of computer and communication systems in an integrated way. This so-called performability modelling becomes especially useful when the system under study can operate partially, which is for instance the case for fault-tolerant computer systems and distributed systems. A prerequisite of a more practical but not less important nature is the availability of software tools to support the modelling techniques and to allow system designers to incorporate the new techniques in the design process of systems. Since performability modelling requires many aspects of a system to be specified, high requirements should be posed on modelling tools. Moreover, these tools should be structured such that the models can be specified at a level that is easy to understand for a system designer, and that the mathematical aspects are hidden as much as possible. The output of the tool should also be such that it can be understood with only limited knowledge of the underlying mathematical model [8], [9].

Performance modelling tools usually have their own textual or graphical specification language which depends largely on the underlying modelling formalism. The different syntax of the tool-specific modelling languages implies that once a tool has been chosen it will be difficult to switch to another one as the model has to be rewritten using a different syntax. On the other hand the solution of the underlying stochastic process is performed in most tools by the same standard numerical algorithms. Starting from these observations the development of MOSEL is based on the following idea: Instead of creating another tool with all the components needed for system description, state space generation, stochastic process derivation, and numerical solution, we focus on the formal system description part and exploit the power of various exisiting and well tested packages for the subsequent stages. MOSEL is a modelling environment which comprises a high-level modelling language that provides a very simple way for system description. In order to reuse existing tools for the system analysis, the environment is equipped with a set of translators which transform the MOSEL model specification into various tool-specific system descriptions [2].

The main features of the MOSEL-environment are the following:

- The modeller inspects the real-world system and generates a high-level system description using the MOSEL specification language. He also specifies the desired performance and reliability measures using the syntax provided by MOSEL. He passes the model to the environment which then performs all following steps without user interaction.
- The MOSEL environment automatically translates the MOSEL model into a tool-specific system description, for example a CSPL-file suitable to serve as input for SPNP.
- The appropriate tool (i.e. SPNP) is invoked by the MOSEL-environment.
- The state space of the model is generated by the tool according to the semantic rules of its modelling formalism out of the static model description.
- The semantic model is mapped onto a stochastic process.
- The stochastic process is solved by one of the standard numerical solution algorithms which are part of the tool. The results of the numerical analysis are saved in a file with a tool-specific structure.
- The MOSEL-environment parses the tool-specific output and generates a result file (sys.res) containing the performance and reliability measures which the user specified in the MOSEL system description. If the modeller requested graphical representation of the results, a second file (sys.igl) is generated by MOSEL [2], [5].

To show an example for using MOSEL we analyze a retrial queueing system. Retrial queues with quasi-random input are recent interest in modeling of magnetic disk memory systems [13], cellular mobile networks [14], and local-area networks with nonpersistent CSMA/CD protocols [11] and star topology [10], [12], [7].

Since in practice some components of the systems are subject to random breakdowns it is of basic importance to study reliability of retrial queues with server breakdowns and repairs because of limited ability of repairs and heavy influence of the breakdowns on the performance measures of the system. For related literature the reader is referred to the works [4], [1], [15] where infinite-source non-reliable retrial queues were treated.

In this paper, finite-source retrial queueing systems with the following assumptions are investigated. Consider a single server queueing system, where the primary calls are generated by K, $1 < K < \infty$ homogeneous sources. The server can be in three states: idle, busy and failed. If the server is idle, it can serve the calls of the sources. Each of the sources can be in three states: free, sending repeated calls and under service. If a source is free at time t it can generate a primary call during interval $(t, t + dt)$ with probability $\lambda dt + o(dt)$. If the server is free at the time of arrival of a call then the call starts to be served immediately, the source moves into the under service state and the server moves into busy state. The service is finished during the interval $(t, t + dt)$ with probability $\mu dt + o(dt)$ if the server is available. If the server is busy at the time of arrival of a call, then the source starts generation of a Poisson flow of repeated calls with rate ν until it finds the server free. After service the source becomes free, and it can generate a new primary call, and the server becomes idle so it can serve a new call. The server can fail during the interval $(t, t + dt)$ with probability $\delta dt + o(dt)$ if it is idle, and with probability $\gamma dt + o(dt)$ if it is busy. If $\delta = 0, \gamma > 0$ or $\delta = \gamma > 0$ *active or independent breakdowns* can be discussed, respectively. If the server fails in busy state, it either *continues servicing* the interrupted call after it has been repaired or the interrupted request *transmitted to the orbit*. The repair time is exponentially distributed with a finite mean $1/\tau$. If the server is failed two different cases can be treated. Namely, *blocked sources* case when all the operations are stopped, that is neither new primary calls nor repeated calls are generated. In the *unblocked (intelligent) sources* case only service is interrupted but all the other operations are continued (primary and repeated calls can be generated). All the times involved in the model are assumed to be mutually independent of each other.

Our main objective is to continue the investigations which were started in [3] but because of page limitations only some results were presented. The mean number of requests staying in the orbit or in the service, overall utilization of the system and the mean response time of calls are displayed as the function of server's failure and repair rates. To achieve this goal MOSEL is used to formulate and solve the problem.

The paper is organized as follows. In Section 2 the full description of the model by the help of the corresponding Markov chain is given. Then,

the main performance and reliability measures are derived that can be obtained using MOSEL tool. In Section 3 several numerical examples are presented and some comments are made. Finally, the paper ends with a Conclusion.

2. The $M/M/1//K$ retrial queue with unreliable server

The system state at time t can be described with the process $X(t) = (Y(t); C(t); N(t))$, where $Y(t) = 0$ if the server is up, $Y(t) = 1$ if the server is failed, $C(t) = 0$ if the server is idle, $C(t) = 1$ if the server is busy, $N(t)$ is the number of sources of repeated calls at time t. Because of the exponentiality of the involved random variables this process is a Markov chain with a finite state space. Since the state space of the process $(X(t), t \geq 0)$ is finite, the process is ergodic for all reasonable values of the rates involved in the model construction, hence from now on we will assume that the system is in the steady state.

We define the stationary probabilities:

$$P(q; r; j) = \lim_{t\to\infty} P(Y(t) = q,\ C(t) = r,\ N(t) = j),$$

$$q = 0, 1,\ \ r = 0, 1,\ \ j = 0, \ldots, K^*,$$

$$\text{where } K^* = \begin{cases} K - 1 & \text{for blocked case,} \\ K - r & \text{for unblocked case.} \end{cases}$$

Knowing these quantities the main performance measures can be obtained as follows:

- *Utilization of the server*

$$U_S = \sum_{j=0}^{K-1} P(0, 1, j).$$

- *Utilization of the repairman*

$$U_R = \sum_{r=0}^{1} \sum_{j=0}^{K^*} P(1, r, j).$$

- *Availability of the server*

$$A_S = \sum_{r=0}^{1} \sum_{j=0}^{K^*} P(0, r, j) = 1 - U_R.$$

- *Mean number of calls staying in the orbit or in service*

$$M = E[N(t) + C(t)] = \sum_{q=0}^{1} \sum_{r=0}^{1} \sum_{j=0}^{K^*} jP(q, r, j) + \sum_{q=0}^{1} \sum_{j=0}^{K-1} P(q, 1, j).$$

- *Utilization of the sources*

$$U_{SO} = \begin{cases} \frac{E[K-C(t)-N(t);Y(t)=0]}{K} & \text{for blocked case,} \\ \frac{K-M}{K} & \text{for unblocked case.} \end{cases}$$

- *Overall utilization*

$$U_O = U_S + KU_{SO} + U_R.$$

- *Mean rate of generation of primary calls*

$$\overline{\lambda} = \begin{cases} \lambda E[K - C(t) - N(t); Y(t) = 0] & \text{for blocked case,} \\ \lambda E[K - C(t) - N(t)] & \text{for unblocked case.} \end{cases}$$

- *Mean response time*

$$E[T] = M/\overline{\lambda}.$$

3. Numerical results

In this section we consider some sample numerical results to illustrate graphically the influence of the non-reliable server on the mean response time, overall utilization of the system and mean number of calls staying in the orbit or in the service. In each case the independent failure is considered and different comparisons are made according to service continuation *(resumed, transmitted)* and system operations *(blocked, unblocked)*.

3.1. The MOSEL implementation. Because of the fact that the state space of the describing Markov chain is very large (especially in the heterogeneous model we would like to investigate later), it is difficult to calculate the system measures in the traditional way of solving the system of steady-state equations. To simplify the procedure and to make our study more usable in practice, we used the software tool MOSEL to formulate the model and to calculate the main performance measures. By the help of MOSEL we can use various performance tools (like SPNP – Stochastic Petri Net Package) to get these measures.

In this section we show the base MOSEL program and the explanation of its main parts without the technical details of programming. This program belongs to the case of continued service after server's repair and request's generation is blocked during the server repairing. It does not contain the picture section, which is needed to generate various graphical representations of the measures. The figures in the next section are automatically generated by the tool with the corresponding picture part. In the MOSEL program we used the following terminology: The server and the sources are referred to as a CPU and terminals, respectively.

```
/* retrialnr-hom-cpu-cont.msl begins */
/*-------------------------------------- Definitions -----------*/
#define NT 6
VAR double prgen;
VAR double prretr;
VAR double prrun;
VAR double cpubreak_idle;
VAR double cpubreak_busy;
VAR double cpurepair;
enum cpu_states {cpu_busy, cpu_idle};
enum cpu_updown {cpu_up, cpu_down};
/*------------------------------------ Node definitions ------*/
NODE busy_terminals[NT] = NT;
NODE retrying_terminals[NT] = 0;
NODE waiting_terminals[1] = 0;
NODE cpu_state[cpu_states] = cpu_idle;
NODE cpu[cpu_updown] = cpu_up;
/*---------------------------------------- Transitions ------*/
IF cpu==cpu_up FROM cpu_idle, busy_terminals
   TO cpu_busy, waiting_terminals W prgen*busy_terminals;
```

```
IF cpu==cpu_up AND cpu_state==cpu_busy FROM busy_terminals
   TO retrying_terminals W prgen*busy_terminals;
IF cpu==cpu_up FROM cpu_idle, retrying_terminals
   TO cpu_busy, waiting_terminals W prretr*retrying_terminals;
IF cpu==cpu_up FROM cpu_busy, waiting_terminals
   TO cpu_idle, busy_terminals W prrun;
IF cpu_state==cpu_idle FROM cpu_up TO cpu_down W cpubreak_idle;
IF cpu_state==cpu_busy FROM cpu_up TO cpu_down W cpubreak_busy;
FROM cpu_down TO cpu_up W cpurepair;
/*------------------------------------------ Results -------*/
RESULT>> if(cpu==cpu_up AND cpu_state==cpu_busy) cpuutil += PROB;
RESULT>> if(cpu==cpu_up) goodcpu += PROB;
RESULT if(cpu==cpu_up) busyterm += (PROB*busy_terminals);
RESULT>> termutil = busyterm / NT;
RESULT>> if(cpu==cpu_up) retravg += (PROB*retrying_terminals);
RESULT if(waiting_terminals>0) waitall += (PROB*waiting_terminals);
RESULT if(retrying_terminals>0)
                              retrall += (PROB*retrying_terminals);
RESULT>> resptime = (retrall + waitall) / NT / (prgen * termutil);
RESULT>> overallutil = cpuutil + busyterm;
/* retrialnr-hom-cpu-cont.msl ended */
```

In the *declaration part* we define the number of terminals (NT), this is the only program code line, that must be modified when modeling larger systems. The terminals have three states: busy (primary call generation), retrying (repeated call generation) and waiting (job service at the CPU). The CPU has two states: idle and busy, and it can be up or failed in both states. We define the three parameters for the terminals: *prgen* denotes the rate of primary call generation, *prretr* references to the rate of repeated call generation and *prrun* denotes the service rate. The *cpubreak_idle, cpubreak_busy* and *cpurepair* variables denote the failure rate in the two CPU states and the repair rate.

The *node part* defines the nodes of the system. Our queueing network contains 5 nodes: one node for the number of busy, retrying and waiting terminals, respectively, and two nodes for the CPU. The CPU is idle and up and all the terminals are busy at the starting time.

The *transition part* describes how the system works. The first transition rule defines the successful primary call generation: the CPU moves from the idle state to busy and the terminal from busy to waiting. The second rule shows an unsuccessful primary call generation: if the CPU is

busy when the call is generated then the terminal moves to state retrying. The third rule handles the case of a successful repeated call generation: the CPU moves from the idle state to busy and the terminal from retrying to waiting. The fourth rule describes the request service at the CPU. The fifth and sixth rules describe the CPU fail in idle and busy state. The last rule shows the CPU repair.

Finally, the *result part* calculates the requested output performance measures.

3.2. Numerical examples. We used the tool SPNP which was able to handle the model with up to 126 sources. In this case, on a computer containing a 1.1 GHz processor and 512 MB RAM, the running time was approximately 1 second.

The results in the reliable case (with very low failure rate and very high repair rate) were validated by the (a little modified) Pascal program for the reliable case given in [6], on pages 272–274. See Table 1 for some test results. The non-reliable case was tested with the non-reliable FIFO model, see Table 2.

In Figures 1–3 we can see the mean response time, the overall utilization of the system and mean number of calls staying in the orbit or in the service for the reliable and the non-reliable retrial system when the server's failure rate increases. In Figures 4–6 the same performance measures are displayed as the function of increasing repair rate. The input parameters are collected in Table 3.

	retrial (cont.)	*retrial (orbit)*	*reliable* [6]
Number of sources:	5	5	5
Request's generation rate:	0.2	0.2	0.2
Service rate:	1	1	1
Retrial rate:	0.3	0.3	0.3
Utilization of the server:	0.5394868123	0.5394867440	0.5394867746
Mean response time:	4.2680691205	4.2680667075	4.2680677918

Table 1. Validations in the reliable case

	retrial (cont.)	*retrial (orbit)*	*non-rel. FIFO*
Number of sources:	3	3	3
Request's generation rate:	0.1	0.1	0.1
Service rate:	1	1	1
Retrial rate:	1e+25	1e+25	–
Server's failure rate:	0.01	0.01	0.01
Server's repair rate:	0.05	0.05	0.05
Utilization of the server:	0.2232796561	0.2232796553	0.2232796452
Mean response time:	1.4360656331	1.4360656261	1.4360655471

Table 2. Validations in the non-reliable case

	K	λ	μ	ν	δ, γ	τ
Figure 1	6	0.8	4	0.5	x axis	0.1
Figure 2	6	0.1	0.5	0.5	x axis	0.1
Figure 3	6	0.1	0.5	0.05	x axis	0.1
Figure 4	6	0.8	4	0.5	0.05	x axis
Figure 5	6	0.05	0.3	0.2	0.05	x axis
Figure 6	6	0.1	0.5	0.05	0.05	x axis

Table 3. Input system parameters

3.3. Comments. In Figure 1, we can see that in the case when the request returns to the orbit at the breakdown of the server, the sources will have always longer response times. Although the difference is not considerable it increase as the failure rate increase. The almost linear increase in $E[T]$ can be explained as follows. In the blocked (non-intelligent) case the failure of the server blocks all the operations and the response time is the sum of the down time of the server, the service and repeated call generation time of the request (which does not change during the failure) thus the failure has a linear effect on this measure. In the intelligent case the difference is only that the sources send repeated calls during the server is unavailable, so this is not an additional time.

In Figure 2 and Figure 5 it is shown how much the overall utilization is higher in the intelligent case with the given parameters. It is clear that

the continued cases have better utilizations, because a request will be at the server when it has been repaired.

In Figure 3 we can see that the mean number of calls staying in the orbit or in service does not depend on the server's failure rate in continuous, non-intelligent case, it coincides with the reliable case. It is because during and after the failure the number of requests in these states remains the same. The almost linear increase in the non-continuous, non-intelligent case can be explained with that if the server failure occurs more often the server will be idle more often after repair until a source repeats his call.

In Figure 4, we can see that if the request returns to the orbit at the breakdown of the server, the sources will have longer response times like in Figure 1. The difference is not considerable too, and as it was expected the curves converge to the reliable case.

In Figure 6, it can be seen that the mean number of calls staying in the orbit or in service does not depend on the server's repair rate in continuous, non-intelligent case, it coincides with the reliable case like in Figure 3. It is true for the non-continuous, non-intelligent case too, which has more requests in the orbit on the average because of the non-continuity.

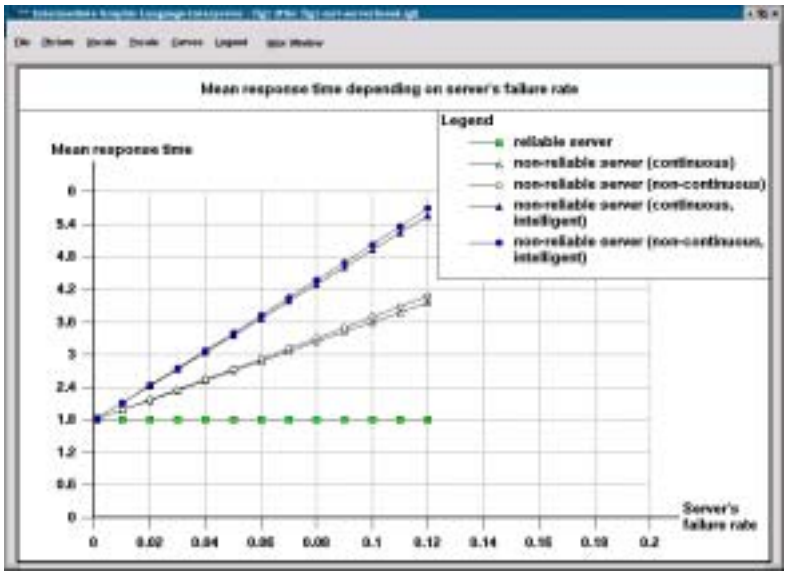

Figure 1

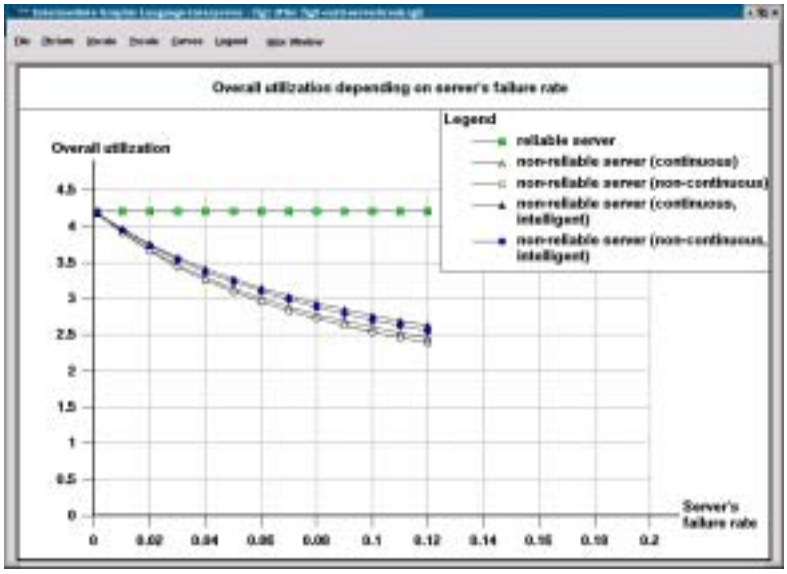

Figure 2

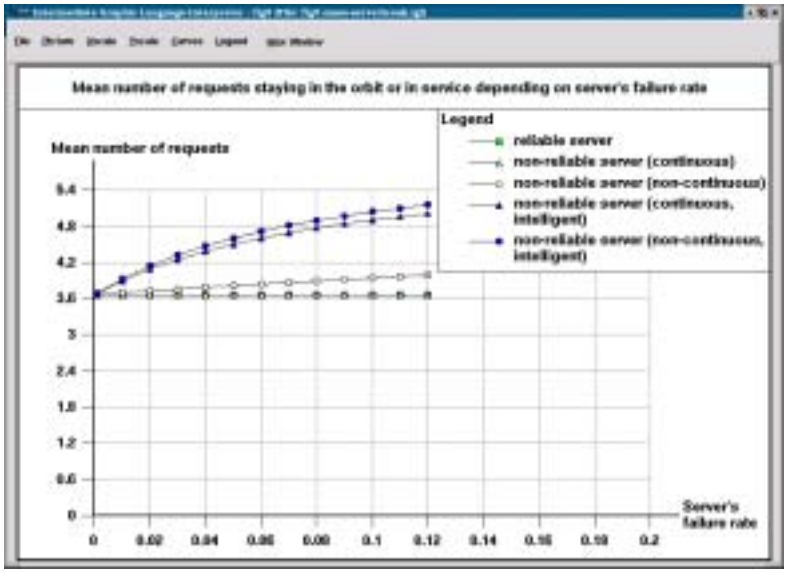

Figure 3

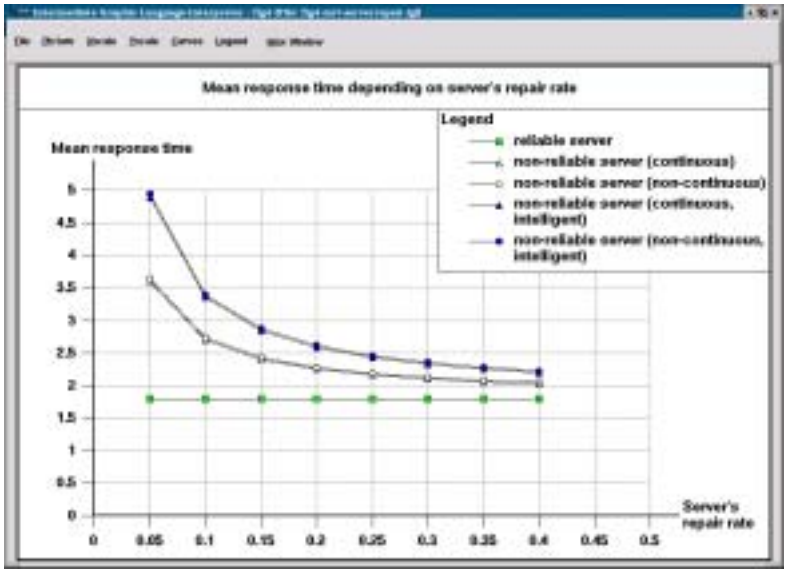

Figure 4

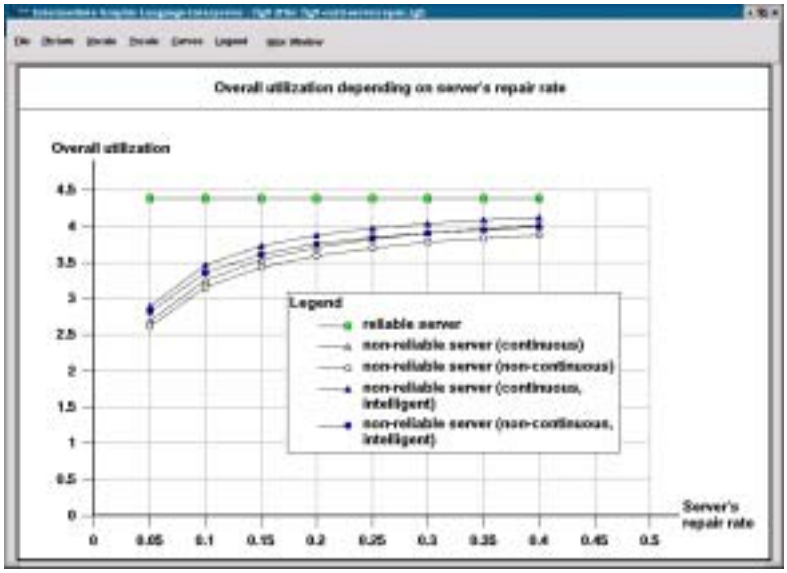

Figure 5

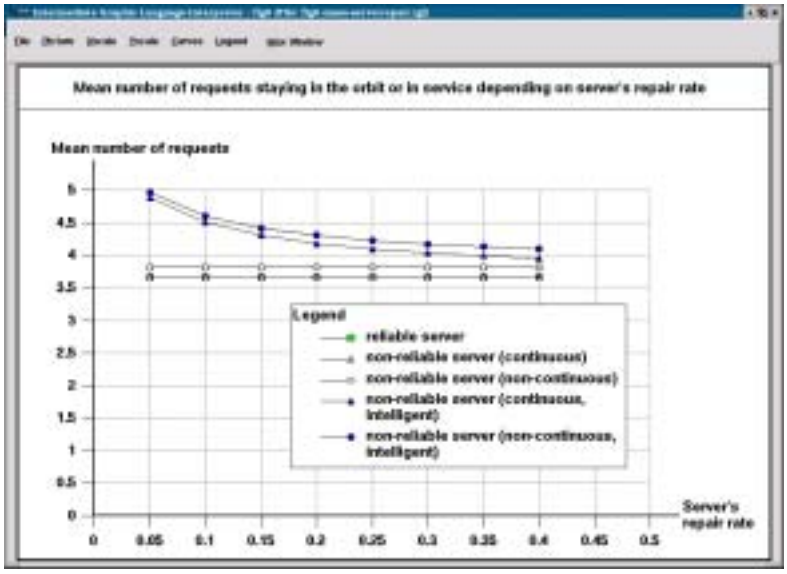

Figure 6

4. Conclusions

In this paper a finite-source homogeneous retrial queueing system is studied with the novelty of the non-reliability of the server. The MOSEL tool was used to formulate and solve the problem, and the main performance and reliability measures were derived and analyzed graphically. Several numerical calculations were performed to show the effect of server's breakdowns and repairs on the mean response times of the calls, on the overall utilization of the system and on the mean number requests staying in the orbit or in service.

References

[1] A. Aissani and J. R. Artalejo, On the single server retrial queue subject to breakdowns, *Queueing Systems Theory and Applications* **30** (1998), 309–321.

[2] K. Al-Begain, J. Barner, G. Bolch and A. I. Zreikat, The performance and reliability modelling language MOSEL and its application, *International Journal of Simulation* **3** (2003), 66–80.

[3] B. Almási, J. Roszik and J. Sztrik, Homogeneous finite-source retrial queues with server subject to breakdowns and repairs (2004) (*to appear in* Computers and Mathematics with Applications).

[4] J. R. Artalejo, New results in retrial queueing systems with breakdown of the servers, *Statistica Neerlandica* **48** (1994), 23–36.

[5] K. Begain, G. Bolch and H. Herold, Practical performance modeling, application of the mosel language, *Kluwer Academic Publisher*, *Boston*, 2001.

[6] G. I. Falin and J. G. C. Templeton, Retrial queues, *Chapman and Hall*, *London*, 1997.

[7] A. Gomez-Corral, Analysis of a single-server retrial queue with quasi-random input and nonpreemptive priority, *Computers and Mathematics with Applications* **43** (2002), 767–782.

[8] B. R. Haverkort and I. G. Niemegeers, Performability modelling tools and techiques, *Performance Evaluation* **25** (1996), 17–40.

[9] B. J. Haverkort, Performance of computer communication systems; A model-based approach, *John Wiley and Sons*, *Chichester*, 1998.

[10] G. K. Janssens, The quasi-random input queueing system with repeated attempts as a model for collision-avoidance star local area network, *IEEE Transactions on Communications* **45** (1997), 360–364.

[11] H. Li and T. Yang, A single server retrial queue with server vacations and a finite number of input sources, *European Journal of Operational Research* **85** (1995), 149–160.

[12] M. K. Mehmet-Ali, J. F. Hayes and A. K. Elhakeem, Traffic analysis of a local area network with star topology, *IEEE Transactions on Communications* **36** (1988), 703–712.

[13] H. Ohmura and Y. Takahashi, An analysis of repeated call model with a finite number of sources, *Electronics and Communications in Japan* **68** (1985), 112–121.

[14] P. Tran-Gia and M. Mandjes, Modeling of customer retrial phenomenon in cellular mobile networks, *IEEE Journal of Selected Areas in Communications* **15** (1997), 1406–1414.

[15] J. Wang, J. Cao and Q. L. Li, Reliability analysis of the retrial queue with server breakdowns and repairs, *Queueing Systems Theory and Applications* **38** (2001), 363–380.

JÁNOS SZTRIK
UNIVERSITY OF DEBRECEN
H-4010 DEBRECEN, P.O. BOX 12
HUNGARY

E-mail: jsztrik@inf.unideb.hu

(Received March 25, 2004)