

# Performance Modeling of Non-homogeneous Unreliable Multi-Server Systems using MOSEL<sup>1</sup>

Aymen I. Zreikat, siayzrei@cip.informatik.uni-erlangen.de  
University of Erlangen, Erlangen, Germany,

Gunter Bolch, bolch@informatik.uni-erlangen.de  
University of Erlangen, Erlangen, Germany,

Janos Sztrik, jsztrik@math.klte.hu  
University of Debrecen, Debrecen, Hungary.

November 7, 2000

## Abstract

In this paper, we introduce a non-homogeneous unreliable multi-server system with Markovian arrival, service, breakdown and repair processes.

First we consider the case with only one queue and different servers and the job assigns to one server. Then we extend this model to more than one queue in which the jobs are assigned to different queues. We assume that our system has different servers with different service times and a job is assigned to a server using the following strategies: FFS (Fastest Free Server) or random selection. FFS strategy means that the job is served by the fastest available server, and if this server is busy then the job goes to the next available server and so on. In the ran-

dom strategy, the job served by one of the free servers which is chosen randomly. In our problem, we consider a general queuing system ( $M/M/n$ ) with a finite number of jobs  $K$  in the whole system. Our system is unreliable; this means that we need to specify the parameters,  $mtbf$  and  $mttr$  (mean time between failures and mean time to repair), and we need to consider the possibility that a server might be up or down at some point of time. The performance modeling of this type of system is done using the programming language *MOSEL (Modeling Specification and Evaluation Language)*, which contains several constructs to describe the system, the results (performance parameters) and the graphical representation.

## 1. Problem Description

The calculation of the performance measure is fairly simple if there is only one server ( $M/M/1$ ), or several servers with identical service time arranged in parallel, and the inter-arrival

---

<sup>1</sup>Acknowledgment: Research is partially supported by German-Hungarian Bilateral Intergovernmental Scientific Cooperation, OMFBDLR No. HUN97/015, OMFKEP grant 0004/99 and Hungarian Scientific Research Fund OTKA-T30685/99.

and service times are exponentially distributed (M/M/n) queuing systems, see for example, [T82, A90]. However, heterogeneous multiple servers often occurs in practice and here the servers have different service times. This situation occurs, for example, when machines of different age or manufacturer are running in parallel. So it is interesting and useful to be able to calculate the performance measures of such systems. In our problem, we consider a queuing system with different number of servers and queues, with the servers have different service times. First, we consider a system with only one queue. We assume that the arrival stream of jobs forms a Poisson process with rate  $\lambda$  and that the service time of jobs at the server are independent and exponentially distributed random variables with a rate  $\mu_i, i = 1 \dots n$ . We further assume that the maximum number of jobs is K, that is, if a job arrives while there are already K jobs in the system, the new job is turned away, such queuing systems are of (M/M/n/K) type.

The problem is divided into two main parts: (i). Typical (M/M/n/K) reliable queuing system, see [BZ2000, Z2000].

(ii). (M/M/n/K) unreliable queuing system. In queuing systems we may find many situations where the terminals or servers are subject to breakdown and repair, see [T82],[S90]. In our model, a server could fail and the failed server could be repaired. It is assumed that the failure and repair times are exponentially distributed with rates depending on the index of the servers and the repair is carried out by a single repairman according to the order of the failure.

In our implementation we use two strategies for selecting the free server:-

- The job is assigned randomly to a free server.
- The Fastest Free Server is selected (FFS).

As we already give priority to every server depending on the service rate (look at Figure 1). We give the first server with higher service rate a higher priority and the one with less service rate one less priority and so on. Then we do not assign the job to the server unless we make sure that this server is working otherwise we have to repair this server and then assign the job to it. So all the time we are interested in the server with higher priority; if it is up then we assign the job to it otherwise we let it go to the repair phase and assign that job to the next server with one less priority. Now, we may face a situation in which the busy server has broken down, then this server has to go to the repair process and the job that we have already assigned to it has to go to the next higher priority server if it is up and free. We apply this strategy to the whole servers alike, depending on their priorities.

So in this case we ignore the quality of the server which is the opposite situation of the homogeneous system (symmetric) in

which we should differentiate between the quality of different servers. That was a description of a straight-forward implementation in the systems that have only one queue but if the number of queues are increased in the system (means more than one queue); which means also that, we have different types of jobs in each queue. Some of the queues give jobs to one server and other give jobs to more than one server. Then in this case we will consider that we have different queues with different classes of jobs in each queue and the jobs inside each queue are in FCFS policy. It is important to mention that in the case where we have only one queue, we need just to care about the priorities of the servers but in the case of several queues in the system, we have to give also priorities to the different queues. Since we have priorities to the servers and priorities to the queues, we shall find a general rule to decide about the relationship between these two priorities (between queues and servers).

By doing different experiments on different graphical representations, we conclude the following general rule:

If we have:

$n$  : the number of servers, then we can decide about the maximum number of queues in that system by the formula:

$$\text{Maximum number of queues} = \left( \sum_{i=1}^n i \right).$$

We also conclude a general rule for giving priorities to different servers and queues, because in our problem some queues provide jobs to different servers on the same time and this cause competition between the queues, so we have to give priorities to different servers and queues.

So please note that we consider different queues with different jobs in each queue but it is also possible to have the situation that there is a FCFS (First Come First Service) policy for the different queues.

In general, with respect to our problem, we will consider the following rules to compute the different priorities to the queues and the servers. Note that we will use these rules in the following programs in all situation where we have many different queues and servers :

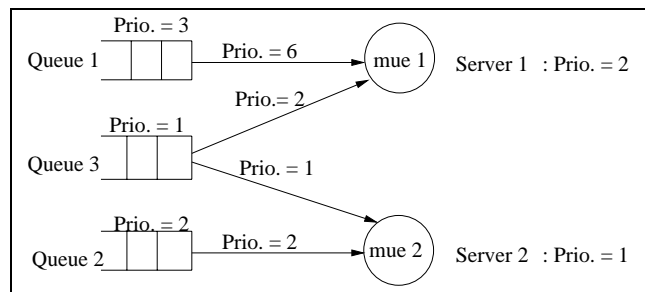


Figure 1: Three Queues 2 Servers (How To Assign Priorities)

- Priority of server  $j = (n - j + 1)$ .
- Priority of queue  $l = (Q - l + 1)$ ,

where  $Q$  is the number of queues in the system.

- We found by experiment that, for every arrow comes from queue to server we can multiply the value of the priority of that queue with the priority of the destination server to get the result priority of the arrow between the two. The graphical representation to explain that is in Figure:1 <sup>2</sup>.
- Please note: **(i)**. The computations for the situation where we have only one queue and many servers; the implementation has been done in a strait-forward way. This means by giving different priorities to the servers and the simulation program assigns jobs to the different servers depending on that priorities. **(ii)**. In the situation where we have many queues and many servers; the implementation has been done by giving priorities to the queues and servers as well. Then we compute the priorities to the arrows going from every queue to the destination server, as we mentioned above. The jobs in the first queue have a higher priority; this means that, these jobs are going to be served first. If there are no jobs in this queue, then the jobs in the queue with one less priority are served next, and so on. Note that the queues that give jobs to the servers directly, have no competition between them. In the simulation program, we have implemented this situation first, then we did the situation where we have competition next. By looking at Figure 1, you will notice the arrows going from the queues to the servers directly without competition and the required priorities to that arrows.

If there are no jobs in the queues that give jobs to the servers directly without competition (queue 1 and queue 2 in Figure 1), then we move one level less priority to this type of queues which give jobs to different servers but in the same time it generates competition with other queues that give jobs to the same server (queue 3 in Figure 1). The jobs in our new queue are served depending also on the priorities given by the arrows going from this queue to the different servers. You will understand the above situation, either by looking to the program code or watching the graphical representation which describes the above situation in Figure 1.

Of course, you have to know that we consider that the jobs come to the queue and they are served by the server depending on the order in which they come to each queue (FCFS policy) ; this means that the policy inside each queue is FCFS policy.

---

<sup>2</sup>An example of 3 queues 2 servers

Following we introduce different problems and the solution of them using MOSEL. Please note that in our implementation we are using MOSEL (MOdeling Specification and Evaluation Language), see, (if you interested), [BBH2000] or [BZ2000], to have an idea about MOSEL; which required the user just to know little about C programming language, because MOSEL is C language plus an additional functionality necessary for the tools we are using (example, SPNP tool).

## 2. One Queue 3 servers Problem

Now for this type of problem, we will consider the two strategies: (FFS, Random), that we have already explained in the Abstract. We will show different results by using the graphical representation provided by MOSEL, and how the performance measures are changing when we give different values for the parameters: (mue1, mue2,...muen) and lambda.

### Strategy 1: FFS (Fastest Free Server)

In all cases, we try to cover the transient solution which is very important to show the behavior of the system over time very clearly. You can watch the results by looking to the related graphical representations provided by MOSEL.

#### Transient Solution

We will show you next our implementation for this type of systems using MOSEL.

#### Mosel Description

This model could be described in *MOSEL* as it is shown in the following listing (**q1s3ffs\_loopfailure.msl**):

```
//PROGRAM   q1s3ffs_loopfailure.msl
//=====
//DEFINITIONS
#define      Q  10 // The number of jobs in the queue
#define      n  3  // The number of servers

#define      mue1 1.9
#define      mue2 1.7
#define      mue3 1.1
#define      lambda 3

#define      mtbf1 500.0 //failure time to server1
#define      mtbf2 100.0 //failure time to server2
#define      mtbf3 10.0  //failure time to server3

#define      mtrr1 10.0 //repaire time to server1
#define      mtrr2 5.0  //repaire time to server2
```

```

#define mtrr3 5.0 //repaire time to server3

//HELP Variables
HELP int K= Q + n;
<1..n> HELP int prio# = n - # +1;

//NODES
enum down_up {down, up}; //enumerated data type
NODE pl[Q];
<1..3> NODE m#[1];
<1..3> NODE state#[down_up] = up;
NODE num[K];

//RULES
FROM TO pl,num W lambda;

<1..3> FROM state#[up] TO state#[down] W 1.0/mtbf#;
<1..3> FROM state#[down] TO state#[up] W 1.0/mtrr#;

<1..n> IF (state#==up) FROM pl TO m# PRIO prio#;

<1..3> FROM m#, num TOE W mue#;

//RESULTS
RESULT>>IF (num==0) prob_system_idl = PROB;
RESULT>>IF (num==K) prob_system_reject = PROB;
RESULT>>Mean_q_length = MEAN pl;
RESULT>> rho1 = UTIL m1;
RESULT>> rho2 = UTIL m2;
RESULT>> rho3 = UTIL m3;
RESULT>> througput = rho1*mue1+rho2*mue2+ rho3*mue3;

//====PICTURE====
PICTURE "Mean Queue Length"
-TITLE "Mean_q_length.....over time"
-FONT utopia -FONTSIZE 22
CURVE TIME Mean_q_length, throug-
put, prob_system_idl,
prob_system_reject
-DIFF THICK,POINT,COLOR,FILL
XSCALE LOOKOUT -GRID
YSCALE LOOKOUT -GRID
//=====

```

## The Results

We start *MOSEL* with the following command:  
**mosel -cst0.5,15,0.5 q1s3ffs\_loopfailure.msl** , in which we use here the option *t* for the transient solution.  
The following result file *q1s3ffs\_loopfailure.res* is provided by this run of the *MOSEL* program:

```

=====
Results provided by the tool 'SPNP'
=====
Constants:
  Q = 10
  n = 3
  mue1 = 1.9
  mue2 = 1.7
  mue3 = 1.1
  lambda = 3
  mtbf1 = 500.0
  mtbf2 = 100.0

```

```

mtbf3 = 10.0
mtrr1 = 10.0
mtrr2 = 5.0
mtrr3 = 5.0
down = 0
up = 1
-----
Results:
K = 13 (Time: 0.5)
.....
= 13 (Time: 15)
prio1 = 3 (Time: 0.5)
.....
= 3 (Time: 15)
prio2 = 2 (Time: 0.5)
.....
= 2 (Time: 15)
prio3 = 1 (Time: 0.5)
.....
= 1 (Time: 15)
prob_system_idl = 0.376902577082 (Time: 0.5)
= 0.247519142478 (Time: 1)
.....
= 0.124825112647 (Time: 15)
prob_system_reject = 1.23327243737e-09 (Time: 0.5)
= 5.98866164356e-07 (Time: 1)
.....
= 0.00179488676598 (Time: 15)
Mean_q_length = 0.0332511182898 (Time: 0.5)
= 0.138010324771 (Time: 1)
.....
= 1.23224007021 (Time: 15)
rho1 = 0.563570015612 (Time: 0.5)
= 0.631413945771 (Time: 1)
.....
= 0.741660982655 (Time: 15)
rho2 = 0.292915413727 (Time: 0.5)
= 0.455775191912 (Time: 1)
.....
= 0.636379905075 (Time: 15)
rho3 = 0.108220521489 (Time: 0.5)
= 0.262912073193 (Time: 1)
.....
= 0.424162882093 (Time: 15)
througput = 1.68778180664 (Time: 0.5)
= 2.26370760373 (Time: 1)
.....
= 2.95758087597 (Time: 15)
Help variables: K=13; prio1=3; prio2=2; prio3=1;
K=13; prio1=3; prio2=2;
.....
prio2=2; prio3=1; K=13; prio1=3
prio2=2; prio3=1;
=====

```

The graphical representation of the file *q1s3ffs\_loopfailure.igl* is also provided by the run of the *MOSEL* program and it is shown in Figure 2. By seeing the above results and looking to the graphical representations in Figure 2, you will notice that the performance measures: (mean queue length, througput, rho1, rho2, rho3) are increasing at the beginning and they start to reach a steady state at the end. We would expect this results from this type of systems since we have just one queue and many servers and the servers have no problem to cover the needs of the queue.

Next, we would like to show the reader how changing of the operating parameters: for example, **lambda** (arrival rate) or **mtbf** (mean time between failure) in the previous program (one queue 3 servers) will affect the results:

for example, in the previous program with  $\lambda = 3$ , we got:

mean queue length at time 1 was equal 0.13

mean queue length at time 15 was equal 1.23

also

throughput at time 1 was 2.26

throughput at time 15 was 2.95

and now we increased the parameter  $\lambda$  (arrival rate) from 3 to 8 and we got the following results:

mean queue length at time 1 equal 2.24

mean queue length at time 15 equal 8.80

also

throughput at time 1 equal 4.18

throughput at time 15 equal 4.29

It is very interesting also to look at the graphical representation that represents the probability that the system is reject in Figure 2 and 3:

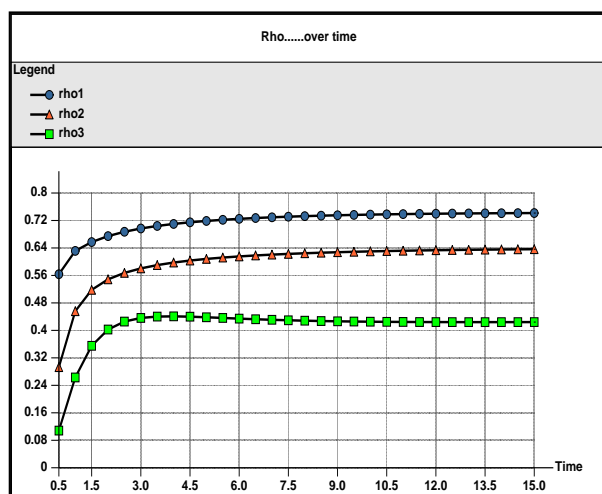
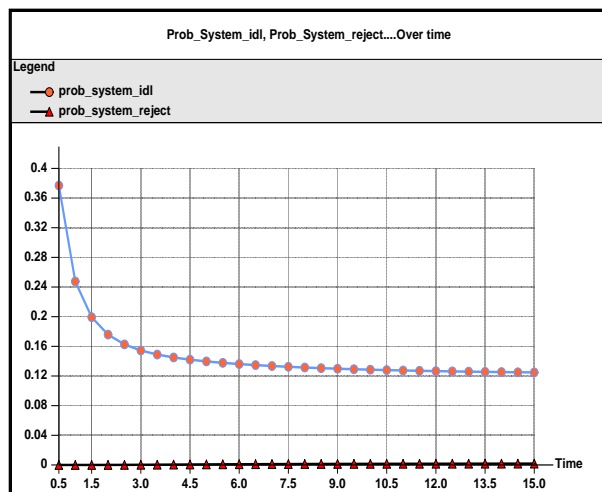
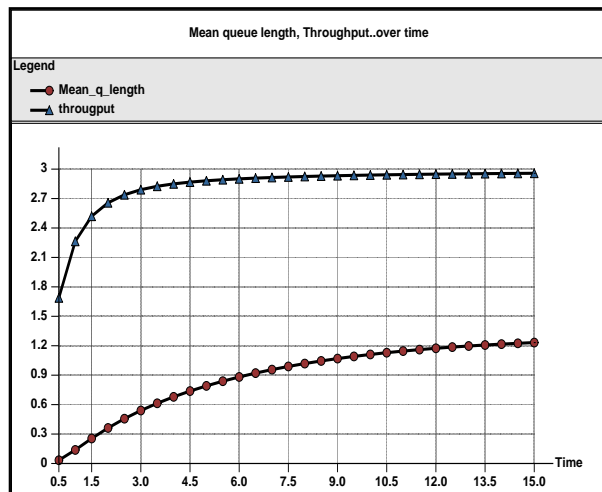
when  $\lambda$  was small the probability was small and goes to a constant value by time. After we increased  $\lambda$ , the probability that the system is reject was also very small in the beginning but it starts to increase by time. So the above results were logical, because in any system if you increase the arrival rate, you should expect that the mean queue length and throughput are also increased dramatically.

You can watch the new results after the above modification of the value of  $\lambda$  by looking at the graphical representation provided by the run of the same MOSEL program with the new value of  $\lambda$  in Figure 3.

- The next modification of the operating parameters is just to try to increase the value of the **mtbf** and see the effect of this increase on the **utilization** of the three servers.

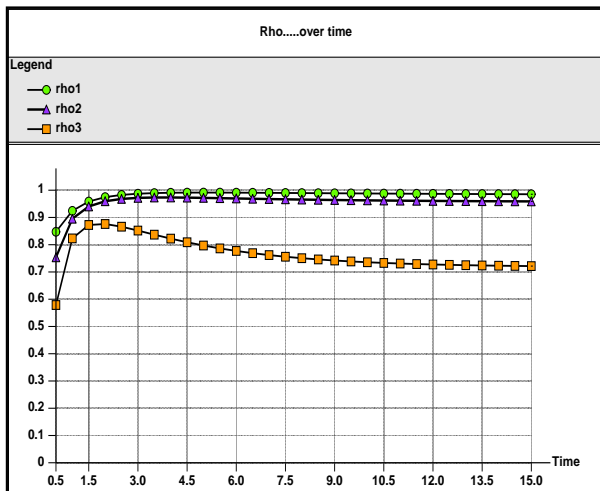
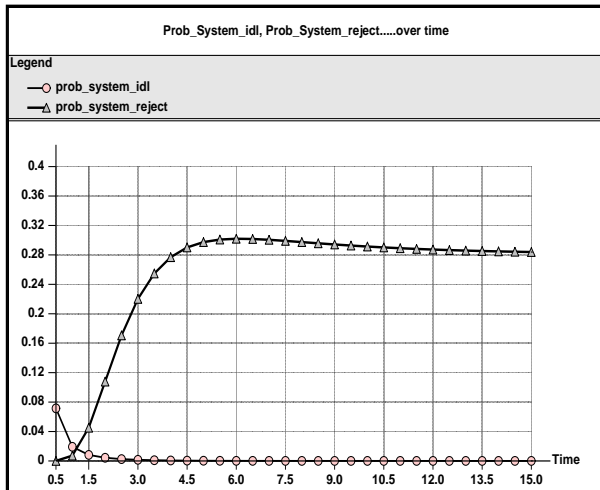
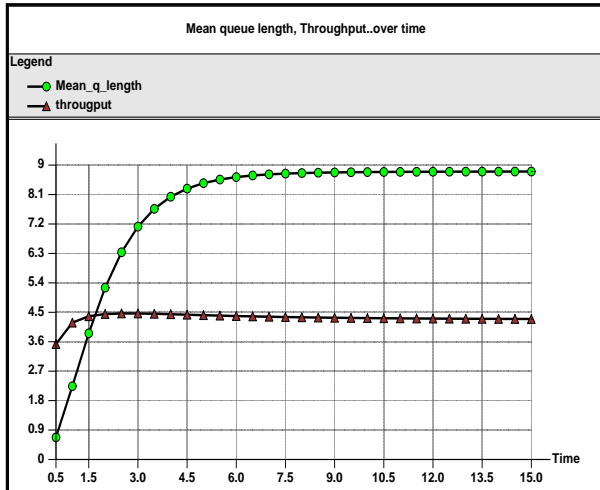
For example, in the previous program with:  $mtbf_1=500$ ,  $mtbf_2=100$ ,  $mtbf_3=10$ , we got the following results for the **rho1**, **rho2**, **rho3**:

for example,  $\rho_1$  at time 15 was equal 0.74



5

Figure 2: (IGL).One Queue 3 Servers (FFS-Loop-Transient (Unreliable System)- $\lambda = 3$ )



rho2 at time 15 was equal 0.63

rho3 at time 15 was equal 0.42

the probability that the server is idle was at time 15 is 0.12

and now, we doubled the values of mtbf1, mtbf2, mtbf3 to be: mtbf1 =1000, mtbf2 =200, mtbf3 =20, and we left the other parameters the same as the original program without any change, then we got the following results for the **utilization**:

for example, rho1 at time 15 is equal 0.73

rho2 at time 15 is equal 0.62

rho3 at time 15 is equal 0.47

and for example, the probability that the system is idle at time 15 is 0.09.

This means that the **utilization** of the first two servers is decreased and the **probability that the system is idle** is increased. This what we should expect from this type of systems. We have also noticed that the **utilization** of the third server is increased. This was an interesting results because we gave priorities to the servers and server3 has the lower priority to get a job and now it has a higher priority to get jobs while the other servers are failed.

Please note that, we tried in the above graphical representations to combine two or more graphs in one graphical representation, which can be done easily by MOSEL, but the result of this combining made the graph unclear, misleading and difficult to understand. So we agree that, it is better to make separation between the performance measures because they have different scales.

Next, we will consider the random case. Actually, with MOSEL it is possible that you can use a loop for abbreviation; so you can replace a set of statements with one statement containing loop.

## Strategy 2: Random

We usually try to show the reader the transient solution, so he or she can watch the behavior of the system over time. Please note that you can also see the steady state solution for these problems using MOSEL. We will show you next the transient solution:

### Transient Solution

### Mosel Description

This model could be described in *MOSEL* as it is shown in the following listing (**q1s3random\_loopfailure.msl**):

Figure 3: (IGL).One Queue 3 Servers (Transient-FFS-Loop-Unreliable System (lambda = 8))

```

//PROGRAM :q1s3random_loopfailure.msl
//=====
//DEFINITIONS
#define mtbf1 500.0 //failure time to server1
#define mtbf2 100.0 //failure time to server2
#define mtbf3 10.0 //failure time to server3

#define mtrr1 10.0 //repaire time to server1
#define mtrr2 5.0 //repaire time to server2
#define mtrr3 5.0 //repaire time to server3

#define Q 10 // The number of jobs in the queue
#define n 3 // The number of servers

#define mue1 1.1
#define mue2 1.9
#define mue3 1.7
#define lambda 2

//HELP Variable
HELP int K = Q + n;

//NODES
enum down_up {down, up}; //enumerated data type

        NODE    p1[Q];
<1..3>  NODE    state#[down_up]= up;
<1..3>  NODE    m#[1];
        NODE    num[K];

//RULES
FROM    TO    p1, num        W        lambda;

<1..3> FROM state#[up]    TO state#[down] W 1.0/mtbf#;
<1..3> FROM state#[down] TO state#[up]   W 1.0/mtrr#;

//CASE1
IF (state1 == up) AND (state2 == up) AND
(state3 == up) AND (m1 == 0) AND (m2 == 0)
AND (m3 == 0)
<1..3> FROM    p1        TO    m#        P        1.0/3.0;

//CASE2
IF (state1 == up) AND (state2 == up) AND
(state3 == down) AND (m1 == 0) AND (m2 == 0)
AND (m3 == 1)
<1..2> FROM    p1        TO    m#        P        1.0/2.0;

//CASE3
IF (state1 == up) AND (state2 == down) AND
(state3 == up) AND (m1 == 0) AND (m2 == 1)
AND (m3 == 0)
<1,3> FROM    p1        TO    m#        P        1.0/2.0;

//CASE4
IF (state1 == down) AND (state2 == up) AND
(state3 == up) AND (m1 == 1) AND (m2 == 0)
AND (m3 == 0)
<2,3> FROM    p1        TO    m#        P        1.0/2.0;

FROM    p1 TO m1 IF (state1 == up) AND (m2 == 1)
AND (m3 == 1);

```

```

FROM    p1 TO m2 IF (state2 == up) AND (m1 == 1)
AND (m3 == 1);
FROM    p1 TO m3 IF (state3 == up) AND (m1 == 1)
AND (m2 == 1);

<1..3> FROM    m#, num        TOE        W        mue#;

//RESULTS
RESULT>>IF (num==0) prob_system_idl    = PROB;
RESULT>>IF (num==K) prob_system_reject = PROB;
RESULT>>Mean_q_length = MEAN p1;
RESULT>> rho1 = UTIL m1;
RESULT>> rho2 = UTIL m2;
RESULT>> rho3 = UTIL m3;
RESULT>> throughput = rho1*mue1+rho2*mue2 +rho3*mue3;

//====PICTURE=====
PICTURE "Mean Queue Length"
-TITLE "prob.system_idl, prob.sys_reject..over time"
-FONT utopia -FONTSIZE 22
CURVE TIME prob_system_idl, prob_system_reject
-DIFF THICK,POINT,COLOR,FILL

XSCALE LOOKOUT -GRID
YSCALE LOOKOUT -GRID
//=====

```

### The results:

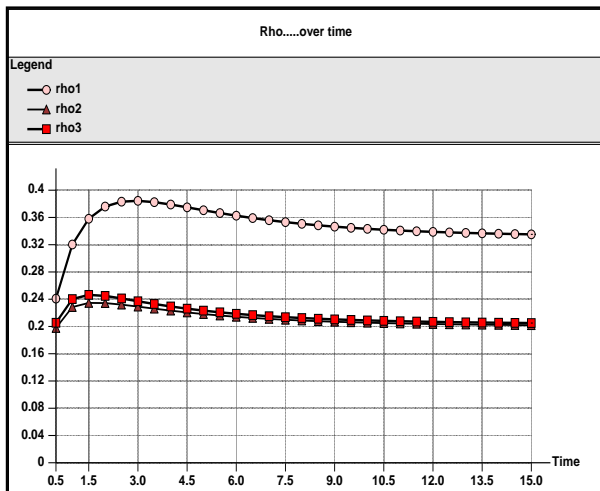
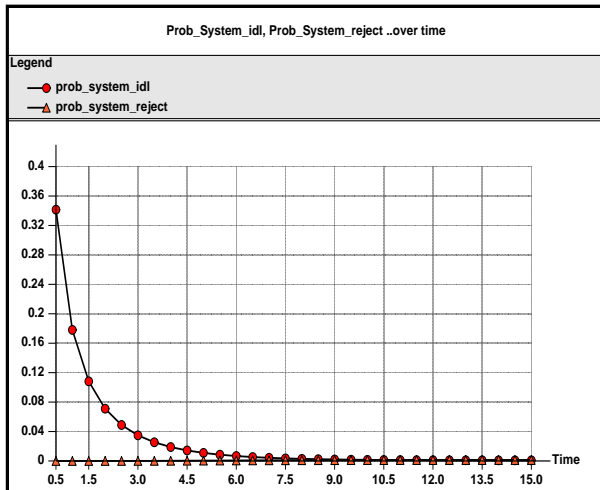
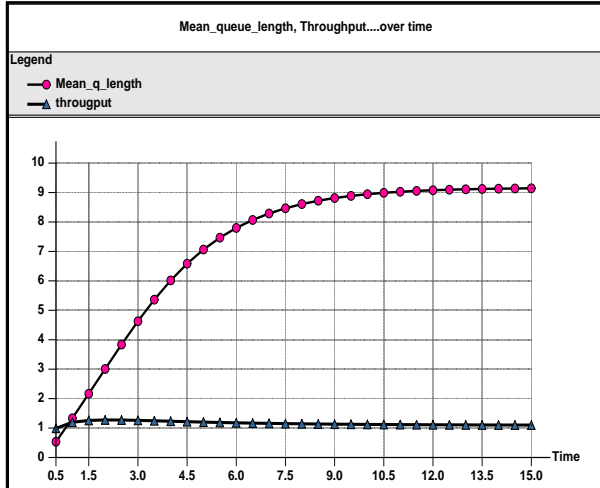
The graphical representation of the file **q1s3random\_loopfailure.igl** is also provided by the run of the **MOSEL** program and it is shown in Figure 4.

### Comments:

If we look at the graphical representations which represent the performance measures: mean queue length, throughput, probability that the system is idle and probability that the system is reject in Figure 2 and Figure 4, you will notice that the behavior is almost the same with respect to the two strategies : FFS and random. On the other hand, if you compare the graphical representations which represent the utilization of the servers: rho1, rho2, rho3, you will notice that there is a clear difference; the utilization of the servers in the FFS strategy is higher than that values in the random strategy. This is expected to this type of systems.

### One Queue n Servers Problem:

Now, the number of servers is not fixed which means that, in MOSEL, we can give any value to the parameter (n) in our program. So just do not define the parameter n in your program, then MOSEL will automatically ask you about the value of n during the execution time of the program. This is very good, in the sense that, we could make the program work in general and in the same time we give the user an opportunity to give the number of servers during the program execution.



Following, we will consider a problems of different type; we will have different number of queues and different number of servers. Please note that, we assumed in our implementations, every queue has a different type of jobs from other queues; which means that we have to assign priorities not just to the servers but also to the queues as well. Following is the case with three queues and two servers.

### 3. Three Queues 2 Servers Problem

Please note that the priorities in the program above are given by an expression and this expression is a function that makes a relationship between the queues priorities and the servers priorities. It multiplies the priority of each queue with the priority of each server and produce the result which is the priority of the arrow going from the queue to server. You can notice this by looking to Figure 1. Actually, we can guess about the behavior of this system before we look at the results. So we expect that in this type of system: the mean number of jobs in the whole system will increase, the mean queue length is also will increase and we expect that the two servers will be utilized and heavily loaded all the time.

Following, will be the FFS and the random solutions for this type of problems. First, we will consider the FFS solution:

#### Strategy 1: FFS (Fastest Free Server)

Please note, that we try to get the transient solution in order to watch the solution over time; which we think is very useful to say something about the system behavior.

#### Transient solution

Also in this type of system, we give you the implementation (code) using MOSEL, but the reader can also say something about the system behavior by looking to graphical representation as well, see (if you are interested) [BBH2000] for more details.

#### Mosel Description

This model could be described in *MOSEL* as it is shown in the following listing (`q3s2ffsfailure.msl`):

Figure 4: (IGL).One Queue 3 Servers (Transient-Random-Loop-Unreliable System)

```

//PROGRAM q3s2ffsfailure.msl
//=====
//DEFINITIONS
#define Q 3 // number of queues
#define n 2 // number of servers
#define l 4 // max number of jobs in every queue

#define mtbf1 500.0 //failure time to server1
#define mtbf2 100.0 //failure time to server2

#define mttr1 10.0 //repaire time to server1
#define mttr2 5.0 //repaire time to server2

<1..Q>#define lambda# #
<1..n>#define mue# 1.<3-#>

//Macro definition
#string rho_mue + rho#*mue#;

//HELP Variables
HELP int K=(Q*1)+n;

<1..n> HELP int a# =(n - #+1) * (Q - #+1);
<1..n> HELP int b# =(n - #+1) * (Q - 3+1);

//NODES
enum down_up {down, up}; //enumerated data type

<1..Q> NODE p#[1];
<1..n> NODE state#[down_up]= up;
<1..n> NODE m#[1];
NODE num[K];

//RULES
<1..Q> FROME TO p#, num W lambda#;

<1..n> FROM state#[up] TO state#[down] W 1.0/mtbf#;
<1..n> FROM state#[down] TO state#[up] W 1.0/mttr#;

<1..n> IF (state#==up) FROM p# TO m# PRIO a#;
<1..n> IF (state#==up) FROM p3 TO m# PRIO b#;

<1..n> FROM m#, num TOE W mue#;

//RESULTS
RESULT>>IF (num == 0) prob_system_idle = PROB;
RESULT>>IF (num == K) prob_system_reject = PROB;
RESULT>>Mean_noj = MEAN num;
<1..n>RESULT>> rho# = UTIL m#;
RESULT>> throughtput = rho1*muel$rho_mue(<2..n>);
//$ is to call Maco
//=====

```

### The results:

The graphical representation of the file `q3s2ffsfailure.igl` is also provided by the run of the *MOSEL* program and it is shown in Figure 5. By looking to the graphical representation in Figure 5, we notice that the probability that the system is idle was high in the beginning and it starts decreasing by the time till it reaches the steady state. On the other hand, the probability that system rejects the job is very low in the beginning and it starts to increase till it reaches again a (steady) stable state. In the above system, we have 3 queues and 2 servers which means that at

some point in time, the servers have to be busy all the time to cop with the requirements of the 3 queues, which is normal to this type of systems.

For the same problem, we will consider next the random strategy, of course, the implementation is somehow similar but the difference is in the way to assign jobs to the servers.

For the random case, we also use the transient solution; which gives us three graphical representations to describe the behavior of the system (see Figure 6).

## Strategy 2: Random

Following is the transient solution:

### Transient Solution

## Mosel Description

This model could be described in *MOSEL* as it is shown in the following listing (`q3s2random_failurenw.msl`):

```

//PROGRAM q3s2random_failurenw.msl
//=====
//DEFINITIONS
#define Q 3 // number of queues
#define n 2 // number of servers
#define l 4 // max number of jobs in every queue

#define mtbf1 500.0 //failure time to server1
#define mtbf2 100.0 //failure time to server2

#define mttr1 10.0 //repaire time to server1
#define mttr2 5.0 //repaire time to server2

<1..Q>#define lambda# #
<1..n>#define mue# 1.<#+3>

#define p1 1.0
#define p2 1.0

//Macro Definition
#string rho_mue + rho#*mue#;

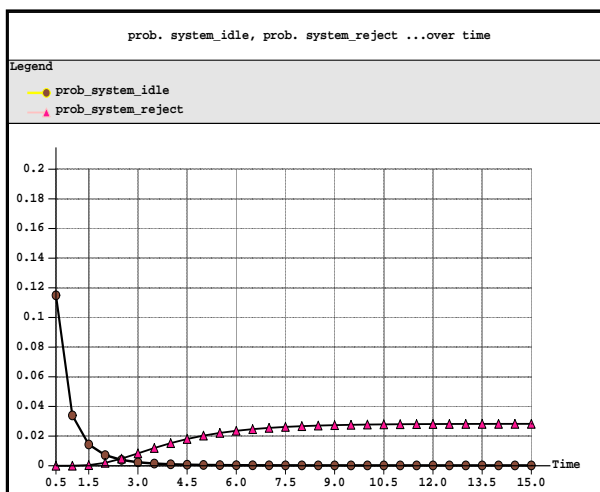
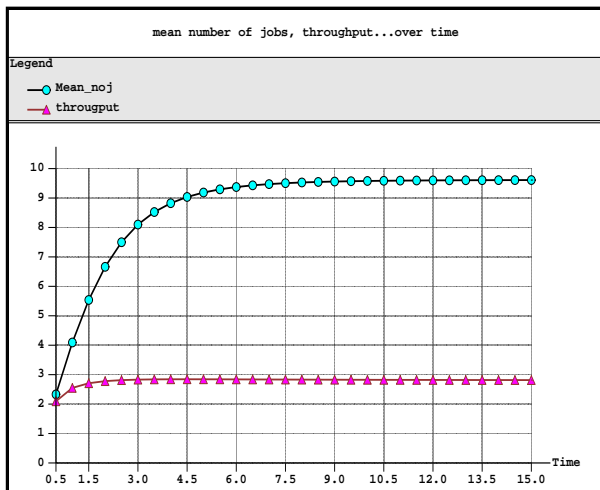
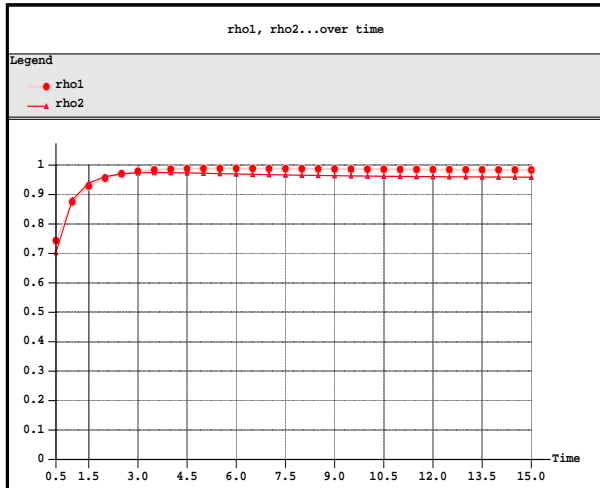
//HELP Variables
HELP int K=(Q*1)+n;

HELP double p31 = 0.7;
HELP double p32 = 1-p31;

//NODES
enum down_up {down, up}; //enumerated data type
<1..n> NODE state#[down_up]= up;
<1..n> NODE m#[1];
<1..Q> NODE Q#[1];
NODE num[K];

//RULES
<1..Q> FROME TO Q#, num W lambda#;

```



```
<1..n> FROM state#[up] TO state#[down] W 1.0/mtbf#;
<1..n> FROM state#[down] TO state#[up] W 1.0/mttr#;
```

```
<1..n> IF (state#==up) FROM Q# TO m# P p#;
<1..n> IF (state#==up) FROM Q3 TO m# P p3#;
```

```
<1..n> FROM m#, num TOE W mue#;
```

```
//RESULTS
RESULT>>IF (num== 0) prob_system_idle = PROB;
RESULT>>IF (num== K) prob_system_reject = PROB;
RESULT>>Mean_noj = MEAN num;
<1..n>RESULT>> rho# = UTIL m#;
RESULT>> throughtput = rho1*mue1$rho_mue(<2..n>);
// $ is to call Macro
```

**The results:**

The three graphical representations for the IGL files are also provided by the run of the *MOSEL* program and they are shown in Figure 6.

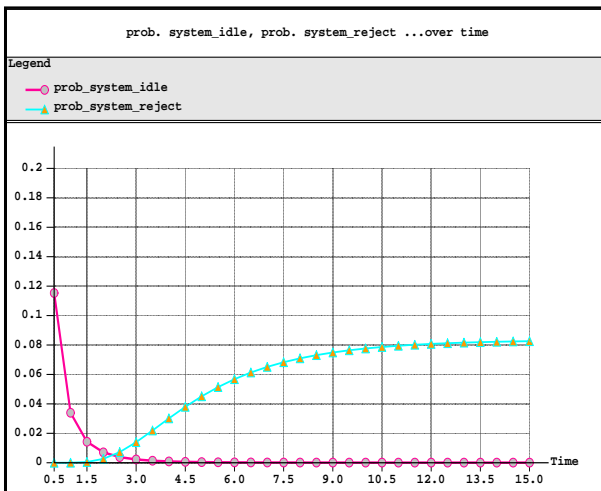
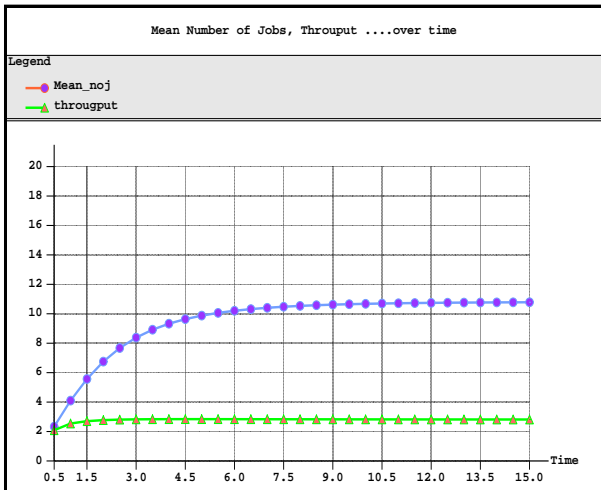
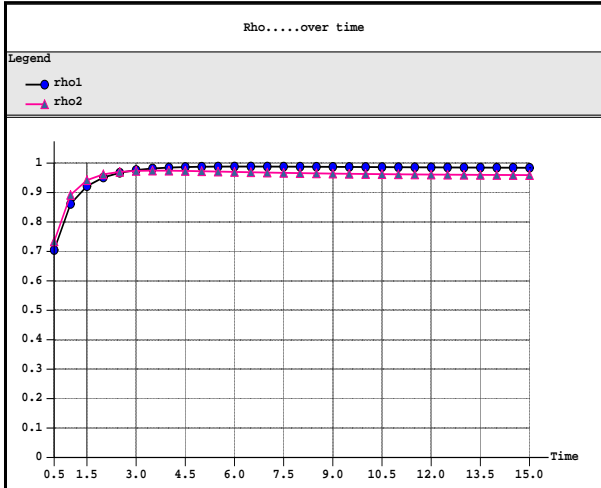
By looking at the graphical representation in Figure 6, we will notice that the *probability that the server idle* and *probability that server reject* are very small, because we have here 3 queues and only 2 servers and that keeps the servers busy all the time. You can also watch the behavior of the other performance measures: rho1, rho2, mean number of jobs and throughput. We should expect these results for this type of systems. You also can find in our daily life many examples about this type of systems (see Applications) for more details.

**Comments:**

We would like to make comparison between the two above strategies: FFS and random for this type of system (three queues two servers) with respect to the performance measures. So if we look at the results in the graphical representations : Figure 5 and Figure 6, we will notice first that, with respect to the performance measures : rho1, rho2, they are almost the same. On the other hand, in the graphical representation that represents the measures: mean number of jobs, throughput; we notice that the mean number of jobs in the whole system is higher in the random case than the FFS case but the throughput is the same. This is clear because in the random case, we choose the available server randomly but in the FFS, we choose just the fastest server, which means that logically we expect that the mean number of jobs in any system that uses the random strategy would be higher than the one that uses FFS strategy.

With respect to the third graphical representation which represents the performance measures: the probability that the system is idle and the probability that the system is reject, the probability that the system is idle is higher in the random strategy than that in the FFS strategy while the probability that the system is reject is almost the same. More important, if you look at

Figure 5: (IGL). Three Queues 2 Servers (Transient-FFS-Unreliable System)



the graphical representation that represents the system with one queue three servers in Figure 2 above, you will notice that the probability that the system rejects the jobs is almost zero, on the other hand, in our case here (3 queues 2 servers), the probability that the system rejects the job was zero at the beginning and it starts to increase till it reaches a stable (steady) state. The explanation of this situation is that, in the case of one queue three servers: it is logically not possible in this type of systems to reach the situation that the system is full and we have to reject the job. One the other hand, in the case of the systems which have three queues and two servers, the possibility that the system will be full is very high and we are not surprised that the probability that the system is full and reject the job is increasing over time. We think that if you understand the meaning of the two strategies, then you should not be surprised from the above results to those types of systems.

### 4. Applications

In the case of one queue and n servers; the jobs come and serve by different types of servers. One example, is a computer network system in which we have many jobs in one queue (FCFS), and these jobs come to different types of printers (servers). Another example of the application is the following situation : some jobs are specialized in just one type of machine (for example LINUX). These jobs are served only by that machine, while the other class of jobs are served by other machine like (UNIX). Some other (third type) jobs could be served either by machine of type one (LINUX) or machine of type two (UNIX). You can find many different examples, for example, in communication and manufacturing systems, too.

### 5. Conclusion

In this paper, it is shown how *MOSEL* can be used to implement non-homogeneous unreliable systems very smoothly. We assumed that our system is unreliable. We have shown the leader how we can solve different problems. We made slight changes on the operating parameters and we have seen how the performance measures changed. We used all the time the transient solution with a useful graphical representations to give the reader an opportunity to watch the system behavior over time. Reliable systems can be solved using *MOSEL* as well, see (if you are interested) [Z2000], for more details.

Figure 6: (IGL).Three Queues 2 Servers (Transient-Random-Unreliable System)

## 6. References

[BGMT98] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. Queuing Networks and Markov Chains, John Wiley & Sons, New York, 1998, 716 pages.

[BBH2000] K. Begain, G. Bolch, H. Herold. Practical Performance Modeling - Application of the MOSEL language, Kluwer Academic Publishers, Boston, 2000, 408 pages.

[BZ2000] G. Bolch, Aymen I. Zreikat, Performance Evaluation of Non-Homogeneous Multi-server systems Using MOSEL, Hamburg, Germany, Presentation ESS 2000 Hamburg.

[S90] J. Sztrik. A recursive solution of a queueing model for a multi-terminal system subject to breakdowns. Performance Evaluation, 11:1-7, 1990.

[T82] K. S. Trivedi. Probability and Statistics with Reliability, Queueing and Computer Science Applications. Prentice-Hall, 1982.

[A90] O. Allen. Probability, Statistics and Queueing Theory with Computer Science Applications. Academic Press, New York, 2nd edition, 1990.

[Z2000] Aymen I. Zreikat. Performance Evaluation of Non-Homogeneous Multi-Server Systems using MOSEL. Master Thesis, Operating System Dep., University Erlangen-Nurnberg, Germany, July, 2000.