Modeling Non-homogeneous Terminal Systems Using MOSEL

B. Almási, almasi@math.klte.hu
Lajos Kossuth University of Debrecen, Hungary,

G. Bolch, bolch@informatik.uni-erlangen.de
Friedrich Alexander University of Erlangen, Deutschland,

J. Sztrik, jsztrik@math.klte.hu
Lajos Kossuth University of Debrecen, Hungary.

September 8, 1999

Abstract

In this paper we consider a stochastic queueing model for the performance evaluation of a real life computer system consisting of n terminals connected with a CPU. A user at terminal i has thinking and processing times depending on index i. First we discuss reliable models, and then unreliable models (i.e. the CPU and the terminals are subject to random breakdowns). The models described here are not new, they were studied earlier by different authors. This paper gives a detailed implementation of the models using MOSEL (MOdeling Specification and Evaluation Language), developed at the University of Erlangen, and it is shown that MOSEL is well suited for modeling such systems. We give some numerical examples, where the most important performance measures (such as utilization, response times, mean queue length) are calculated, to see what is the difference between the results of MOSEL and the ones published earlier. In the appendices we publish the source codes of the MOSEL programs mentioned in this paper.

1 Introduction - the structure of a MOSEL program.

In this section we give a short overview on the structure of a MOSEL program, which is useful for studying the MOSEL implementation of the different models discussed later in this paper. For a more detailed description of MOSEL see [BOLCH99].
The MOSEL programs discussed in this paper consist of four parts: the declarations, the node definitions, the transition rules and the results. We can put comments into the MOSEL program enclosed in /* ... */ (like in C). In all parts we can use shortcuts, to make the program shorter. In the shortcut we enclose the indices (or an interval of indices) in < ... >, then in the same line we can reference to the actual index value with a # character (see below the examples).

1.1 The declaration part.

In this part we can define constants, declare variables and enumerated types. The enumerated types can be used to describe the set of the states of a node. Macro definition is also possible using the string keyword. Here is a small example:

```
/*================ Declaration part begins ==================*/

/*======= Definition of a constant MAXNT with value 9 ========*/
#define MAXNT 3
```

[1]Acknowledgment: The authors are very grateful to Helmut Herold for his helpful comments. Research is partially supported by German-Hungarian Bilateral Intergovernmental Scientific Cooperation, OMFB-DLR No. 015-97, and OM-FKFP grant 0004/99.

```
/*========== Definition of a macro NT with value 3 ===========*/
#string  NT  3:

/*== Definition of input variables prgen1, prgen2, prgen3 ===*/
/*============================ and prrun1, prrun2, prrun3 ===*/
/*= Here we use shortcuts, to make the declarations shorter =*/
<1..$NT> VAR double prgen#;
<1..$NT> VAR double prrun#;

/*============= Here is an enum type example ================*/
enum  term_states { busy, waiting };
```

1.2 The node part.

In this part we define the nodes for the system. We can use the constants and enum types defined in the declaration part. We can also give initial values for the nodes, as you can see in the following examples:

```
/*================ The NODE part begins ======================*/

/*== Definition of nodes named term with states term_states ==*/
<1..$NT> NODE term#[term_states] = busy;

/*========= Definiton of a node with capacity NT =============*/
NODE que[$NT] = 0;
```

1.3 The transition rule part.

This is the most important part of the MOSEL program, which describes the system's behavior using FROM ... TO style rules. Please refer to [BOLCH99] for further details. Here is a small example of the rule part:

```
/*================= The rule part begins ====================*/

/* If the state of term1 is busy, then we can move a job to the
   CPU with rate prgen1, this sets the state of term1 to waiting. */
if term1 == busy {
              FROM  term1[busy]   TO  cpu, term1[waiting]  WITH prgen1;
}


/* From the queue we we can service the job of term1 with rate prrun1,
   and set the state of term1 to busy. MOSEL automatically checks
   the assumption if term1 == waiting and CPU > 0. */
FROM  cpu, term1[waiting]  TO   term1[busy]  WITH prrun1;
```

1.4 The result part.

This section calculates the output results. The results are specified by equations, giving the name of the "output variable" on the left side, and the formula on the right side. On the right side we can use the word PROB to refer to the steady-state probability of the given state:

```
/*================ The result part begins ===================*/

/*========  If a terminal is busy, then it is utilized =======*/
<1..$NT> RESULT>> if ( term# == busy ) term#util += PROB;
```

```
/*==== Calculate the average length of the queue of the CPU ====*/
RESULT>> quelength = MEAN cpu;
```

1.5 Starting MOSEL.

We can start MOSEL with the command
`mosel -cs mosel_file`
where `mosel_file` refers to the source file of the MOSEL program. Mosel creates an SPNP input file (see [CIARDO91], [TRIVEDI91]), then calls the C compiler, to compile it and link the SPNP objects to it. Finally the new executable is automatically started, and the results are written into the file `mosel_file.res` .

The compilation can also be done manually using the following syntax:
`mosel -c mosel_file`
`spnp mosel_file`
In this case we get an executable file `mosel_file.spn`, which can be used later to start the analysis (without MOSEL nor SPNP). If we used input variables in the MOSEL program, to get the required model parameters, then the executable can be used alone to analyse such a systems later. Starting the executable with the command `mosel_file.spn outfile` we will get the results in the file `outfile.out`.

2 Modeling reliable non-homogeneous terminal systems.

2.1 The mathematical model.

This section deals with a terminal system consisting of n terminals connected with a Central Processing Unit (CPU). The model is a closed queueing network with 2 (multiple server) service stations (nodes) and finite number n of jobs. The model was discussed in detail in [CSIGE82]. The first service station is the processor, consisting of only one server (the CPU itself), where jobs from the terminals may suffer from queueing delay. We consider three service disciplines at the CPU: FIFO, Priority Processor Sharing (PPS), which includes the Processor Sharing (PS) discipline too (see [SZTRIK86]), and the Polling discipline (see [ALMASI99]).

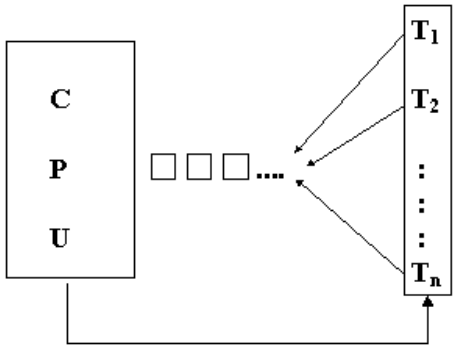

Figure 1.

The second service station is a collection of n terminals (i.e. multiple server station). At the terminals there is no queueing delay for the jobs (as the number of terminals is equal to the number of jobs). A user at terminal i has thinking (i.e. program generation) times, and processing (i.e. program running) times depending on index i. Let us denote by λ_i, μ_i, w_i the parameters of the exponentially distributed thinking, processing times and weight for terminal i, i=1,...,n, respectively. We assume, that each user generates only one job at a time, and he waits until the CPU services it. The random variables are assumed to be exponentially distributed and independent of each other. The system state at time t can be described with the following stochastic process

$$M(t) \quad = (i_1, .., i_k),$$

where $(i_1, ..., i_k)$ denotes the indices of the jobs residing at the CPU at time t. Depending on the service discipline the random variable M(t) gives the order of service by the CPU, too. It can easily be seen

that the stochastic process M(t) is a Markov chain having a rather complex, and large state space (see [CSIGE82]). Let us denote the steady-state distribution of $(M(t), t \geq 0)$ by

$$p(i_1 \dots i_k) = \lim_{t\to\infty} p(M(t) = (i_1, \dots, i_k))$$

Furthermore, let us denote by $p(k)$, $(k = 0, 1, ..., n)$ the steady-state probability, that k jobs are at the CPU. Knowing these probabilities the main performance measures can be calculated with a simple sum operation over the steady-state probabilities, as follows (see [CSIGE82], [SZTRIK86], [ALMASI99]):

(i) Mean number of jobs residing at the CPU

$$\overline{n}_j = \sum_{k=0}^{n} kp(k).$$

(ii) Mean number of busy terminals

$$\overline{n}_b = \sum_{k=0}^{n} (n-k)p(k).$$

(iii) Utilization of the CPU

$$U_{CPU} = \sum_{k=1}^{n} p(k).$$

(iv) Probability, that the job of the terminal i is in the queue of the CPU

$$Q_i = \sum_{k=1}^{n} \sum_{r=1}^{k} \sum_{i_1,\dots,i_k} \delta(i, i_r)p(i_1, \dots, i_k).$$

(v) Utilization of terminal i, i=1,...,n

$$U_i = 1 - Q_i,$$

(vi) Expected response time of jobs for terminal i, i=1,...,n (see [ALMASI96])

$$T_i = \frac{Q_i}{\lambda_i U_i}$$

2.2 The MOSEL implementation of the model.

In this subsection we discuss two different MOSEL implementations of the above model. We discuss the case of 3 terminals and FIFO service discipline at the CPU. Only the most important parts of the MOSEL programs are considered here, the full source of the programs for 3 terminals using FIFO, PPS and POLLING service rules can be found in the Appendices.

2.2.1 Implementation using a long transition rule part.

In this implementation we define an enum type named que_states, which consists of values written in S_ij form. The value S_ij shows, that there are 2 jobs at the CPU from terminal i and j, in this order, and the job of terminal i is processed.
We can describe the mathematical model introduced above by defining a node with states in que_state. We define two other nodes too, to help to calculate performance characteristics. The first one is the cpuque, which shows the number of the jobs at the CPU; the second one is the terminal1, terminal2, terminal3 group of nodes (defined with a shortcut), to show the state of the terminals (busy or waiting):

```
/*============ Declarations and definitions ============*/

<1..3> VAR double prgen# ;
<1..3> VAR double prrun# ;


/*================= The states ========================*/
enum  term_states { busy, waiting };
```

```
enum  que_states {
          S_0,          S_1,          S_2,
          S_3,          S_12,         S_13,
          S_21,         S_23,         S_31,
          S_32,         S_123,        S_132,
          S_213,        S_231,        S_312,
          S_321     };

/*================== The nodes ======================*/
        NODE system[que_states]=S_0;
<1..3>  NODE terminal#[term_states]=busy;
        NODE cpuque[3]=0;

/*======== Declarations and definitions ended =======*/
```

The transition rules of the MOSEL program can be defined according to the following form:

```
/*================== Transition rules ================*/

IF  system == S_2   {
      FROM  cpuque,terminal2[waiting],S_2 WITH prrun2  TO  terminal2[busy],S_0;
<1,3> FROM  terminal#[busy],S_2 WITH prgen#  TO  cpuque,terminal#[waiting],S_21;
 }
/*================ Transition rules ended ============*/
```

The rule is valid, when the system is in the state S_2 (i.e. there is one job at the CPU from terminal 2). The first line says, that a transition is possible from this state by servicing the job. The second says, that for the other terminals (which are not in the queue, i.e. terminal 1 and 3) a job generation is possible. The problem of the implementation is, that we have to create these rules for all the states. Although the structure of the transitions is very simple, the MOSEL program becomes very large. We can generate the MOSEL source by a small C program to avoid the typing mistakes (see in Appendix H).
The result part of the MOSEL program looks like this:

```
/*======================== Results ======================*/
RESULT>> if ( cpuque != 0 ) cpuutil += PROB;
RESULT>> cpuquelength = MEAN cpuque;
<1..3> RESULT>> if (terminal# == busy) termutil# += PROB;
<1..3> RESULT if (terminal# == waiting) termqueing# += PROB;
<1..3> RESULT>> resptime# = termqueing# / ( prgen# * termutil# );
/*====================== Results ended ==================*/
```

In the result part we calculate a new value *termqueing*, but it is not printed out (the >> sign missing from the end of the RESULT word), it is only used to calculate the response time. The variable cpuutil shows the utilization of the CPU, similarly termutil1, termutuil2, termutil3 denotes the utilization of the appropriate terminal. The variable cpuquelength shows the average length of the queue of the CPU. The variable resptime denotes the average response time for the given terminal according to [ALMASI96]. The full version of the MOSEL program can be found in Appendix A.

2.2.2 The short and more efficient implementation.

Thanks to the helpful comments of Helmut Herold, we could achive a big simplification to the MOSEL program introduced above. Let us see the definitions and node declarations of the short program:

```
/*------------------------------------------------ Definitions -----------*/
#define  NT   3         // change it to 4, 5, 6, 7, ...

/*====================== No changes required below ======================*/
```

```
<1..NT> VAR double prgen# ;
<1..NT> VAR double prrun# ;

/*------------------------------------------------ Node definitions -------*/
<1..NT>  NODE terminal#[1] = 1;
         NODE cpuque[NT]  = 0;
<1..NT>  NODE p#[NT]      = 0;
```

The last line declares a vector of nodes, this vector can be used to describe the queue at the CPU. The node p1 contains the index of the first job in the queue, p2 contains the index of the second job etc.
Let us start explaining the transition rules of the implementation one by one:

```
/*------------------------------------------------------ Transitions -------*/
<1..NT><NT>                 FROM terminal<#1>   TO cpuque, p<#2>(<#1>)
                                W prgen<#1>;
```

This rule is the rule of program generation. MOSEL automatically checks the assumption if $terminal < \#1 >== 1$ (as a property of the FROM keyword). The expression $p < \#2 > (< \#1 >)$ says, that we put the index of the terminal (i.e. #1) to the last place (i.e. NT-th place) of the vector p.

```
<1..NT>       IF p1==#      FROM cpuque, p1(#)  TO terminal#
                                W prrun#   ;
```

This rule is the job service rule at the CPU. We also clear the terminal index from p1, so after the rule p1=0.

```
<2..NT>       IF p<#-1>==0  FROM p#(p#) TO p<#-1>(p#);
```

This rule contains a non-timed transition, to skip the zero values, and put the job indices into the beginning of the vector p. It is necessary when a new job has been put into the end of the queue vector p, or when a job was serviced and canceled from the beginning of the vector.

```
/*------------------------------------------------------------ Results -------*/
        RESULT>> cpuutil        = UTIL cpuque                          ;
        RESULT>> cpuquelength   = MEAN cpuque                          ;
<1..NT> RESULT>> termutil#      = UTIL terminal#                       ;
<1..NT> RESULT   termqueueing#  = 1-termutil#                          ;
<1..NT> RESULT>> resptime#      = termqueueing# / (prgen# * termutil#);
```

This result part is very similar to the longer version.
The PPS and POLLING versions of the MOSEL programs can be build up using a similar way. The MOSEL sources for FIFO, PPS, POLLING service rules can be found in Appendix B, C, D. There are two important changes at the PPS case: we have to use a C function to calculate the divider of the job service transition rate, and after setting the job index to the last place of the vector we have to skip not only the zero, but the greater elements too.

2.2.3 Numerical Results.

In this chapter we consider some numerical examples - studied earlier by different authors - to check the programs.
We tried the short and the long MOSEL programs too. Concerning the output results we found no differences, both method produced exactly the same numerical results.
Comparing the running times (without the compilation time) of the two versions, we can see in Table 1, that the short method is more efficient, and can be used for $n = 6, 7$ too (for the long program $n = 5$ was the largest n value, that could be used). The measurement was made on a SUN ULTRA 10 (300 MHz CPU clock, 512 MB RAM)

	$n=3$	$n=4$	$n=5$	$n=6$	$n=7$
Long version	0.03	0.11	2.31	n.a.	n.a.
Short version	0.03	0.10	0.54	4.96	166.94

Table 1. Running times (in seconds).

Case 1.
The following example was described in [CSIGE82] for FIFO and PS and in[ALMASI99] for POLLING service discipline at the CPU. We can see, that our calculation is equivalent to the earliers, differences appear only in the 3-rd decimal digit:

Input parameters:
$n = 4$

i	λ_i	μ_i
1	0.3000	0.6000
2	0.4000	0.7000
3	0.2000	0.5000
4	0.5000	0.9000

Performance measures:

	FIFO	PS	POLLING
n_j	2.185934	2.195422	2.184688
U_{CPU}	0.903398	0.906375	0.903018
U_1	0.469156	0.451775	0.471194
U_2	0.416263	0.423391	0.414826
U_3	0.546164	0.500074	0.552183
U_4	0.382481	0.429335	0.377107
T_1	3.771613	4.044955	3.740888
T_2	3.505817	3.404702	3.526618
T_3	4.154753	4.998512	4.054961
T_4	3.229007	2.658358	3.303534

Table 2. An example for $n = 4$.

Case 2.
The following example was introduced by J. Sztrik in 1986 for PPS service discipline using 3 terminals. The results of MOSEL are the same to the ones published in [SZTRIK86].

Input parameters:
$n = 3$

i	λ_i	μ_i	w_i
1	0.2000	0.4000	1.0
2	0.2000	0.6000	5.0
3	0.2000	0.8000	125.0

Performance measures:

U_{CPU}	n_j
0.675675	1.028136

i	U_i	T_i
1	0.508578	4.831325
2	0.666782	2.498694
3	0.796502	1.277448

Table 3. An example for $n = 3$ using PPS discipline.

3 Modeling non-reliable non-homogeneous terminal systems.

3.1 Modifications to the mathematical model.

The model introduced in 2.1 is the starting point of the non-reliable models, discussed in this section. We add the following modifications to the basic model: There is a new service station (node), named repairman. We assume, that the busy equipments (terminals, CPU) are subject to random breakdowns, so giving duties to the repairman. The random working and repair times of the terminals are exponentially distributed with mean depending on the terminal index (non-homogeneous breakdowns). Furthermore we assume, that the CPU is responsible for the system's work, i.e. the service stops at the terminals, and at the CPU, when the CPU is down. The repairman gives preemptive priority to the CPU failure, and follows FIFO discipline for the terminal breakdowns.

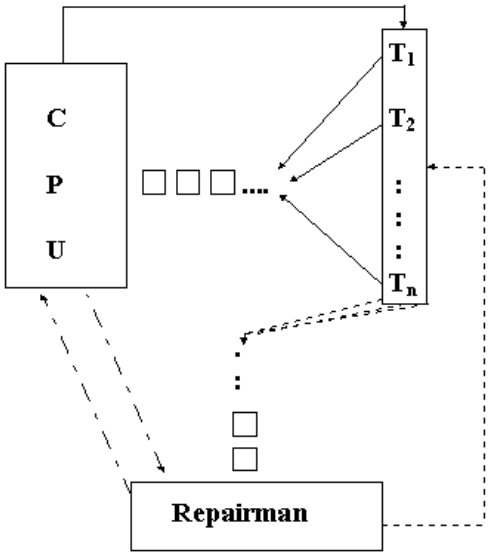

Figure 2.

Let us denote by λ_i, μ_i, γ_i, τ_i, w_i the parameters of the exponentially distributed thinking, processing, operating, repair times and weight for terminal i, i=1,...,n, respectively. Similarly, let α, β denote the failure and repair rate of the CPU, respectively.
To deal with the problem we have to introduce the following random variables:

$$\begin{aligned} X(t) &= \begin{cases} 1, & \text{if the operating system is failed at time } t, \\ 0, & \text{otherwise,} \end{cases} \\ Y(t) &= \text{the failed terminals' indices at time } t \text{ in order of their failure,} \\ &\quad \text{or 0 if there is no failed terminal,} \\ Z(t) &= \text{the indices of the jobs residing at the CPU at time } t, \\ &\quad \text{or 0 if the CPU is idle.} \end{aligned}$$

Depending on the service discipline the random variable Z(t) gives the order of service by the CPU, too. It can easily be seen that the stochastic process $M(t) = (X(t), Y(t), Z(t))$ is a Markov chain having a rather complex, and large state space. To get its the steady-state probabilities an efficient recursive computational method has been introduced and used for different service rules mentioned earlier, c.f. [ALMASI92], [SZTRIK90]. Let us denote the steady-state distribution of $(M(t), t \geq 0)$ by

$$\begin{aligned} &p(q; i_1 \dots i_k; j_1, \dots, j_s) = \\ &= \lim_{t\to\infty} p(X(t) = q; Y(t) = i_1, \dots, i_k; Z(t) = j_1, \dots, j_s) \end{aligned}$$

Furthermore, let us denote by $p(q, k, s)$ the steady-state probability that the operating system is in state q, k terminals are failed and s jobs are at the CPU.
Knowing these probabilities the main performance measures can be obtained as follows:
(i) Mean number of jobs residing at the CPU

$$\overline{n}_j = \sum_{i=0}^{1} \sum_{k=0}^{n} \sum_{s=0}^{n-k} sp(i, k, s).$$

(ii) Mean number of good terminals

$$\overline{n}_g = n - \sum_{i=0}^{1} \sum_{k=0}^{n} \sum_{s=0}^{n-k} kp(i,k,s).$$

(iii) Average number of busy terminals

$$\overline{n}_b = \sum_{k=0}^{n} \sum_{s=0}^{n-k} (n-k-s)p(0,k,s).$$

(iv) Utilization of the repairman

$$U_r = \sum_{k=0}^{n} \sum_{s=0}^{n-k} p(1,k,s) + \sum_{k=1}^{n} \sum_{s=0}^{n-k} p(0,k,s).$$

(v) Utilization of the CPU

$$U_{CPU} = \sum_{k=0}^{n-1} \sum_{s=1}^{n-k} p(0,k,s).$$

(vi) Utilization of terminal i, i=1,...,n

$$U_i = \sum_{k=0}^{n} \sum_{s=0}^{n-k} \sum_{i_1,\ldots,i_k} \sum_{j_1,\ldots,j_s} (\prod_{r=1}^{k} \prod_{v=1}^{s} (1 - \delta(i,i_r) - \delta(i,j_v))) \quad p(0; i_1,\ldots,i_k; j_1,\ldots,j_s).$$

(vii) Expected response time of jobs for terminal i

$$T_i = \frac{Q_i}{\lambda_i U_i}$$

where $\delta(i,j) = \begin{cases} 1, \text{ if i=j,} \\ 0, \text{otherwise,} \end{cases}$ and Q_i denotes the probability of staying at the CPU for terminal i, namely

$$Q_i = \sum_{q=0}^{1} \sum_{k=0}^{n-1} \sum_{s=1}^{n-k} \sum_{r=1}^{s} \sum_{i_1,\ldots,i_k} \sum_{j_1,\ldots,j_s} \delta(i,j_r)p(q; i_1,\ldots,i_k; j_1,\ldots,j_s).$$

3.2 The MOSEL implementation of the non-reliable model.

In this subsection we discuss the MOSEL implementation of the non-reliable model using PPS discipline at the CPU. The full source of the MOSEL programs for FIFO, PPS, POLLING service rules can be found in Appendix E, F, G.

```
/*== Non-reliable terminal-system PPS service for NT pieces of terminals ===*/
/*------------------------------------------------ Macro definition -------*/
#string  w_assign  w[#]=weight#; :
#string  a_assign  a[#]=p#;      :

#define  NT   3         // change it to 4, 5, 6, 7, ... MAXNT
/*======================= No changes required below ========================*/
#define  MAXNT 15
<NT>    #define NT1 <#+1>

        VAR double cpubreak;
        VAR double cpurepair;
<1..NT> VAR double termbreak# ;
<1..NT> VAR double termrepair# ;
<1..NT> VAR double prgen# ;
<1..NT> VAR double prrun# ;
<1..NT> VAR double weight# ;

enum term_states {busy, waiting, failed};
enum cpu_states {up, down};
```

The definitions are similar to the ones in the reliable model. Here we define two enum types to describe the states of the CPU and the terminals. The two macros w_assign and a_assign will be used later to put the weights values and job indices into arrays.

```
/*------------------------------------------- Function definitions -------*/
double divider(void)
{       int i,a[MAXNT+1];
        double sum, w[MAXNT+1];

/*======== Put the weights and the CPU queue indices into arrays =========*/
        $w_assign(<1..NT>)
        $a_assign(<1..NT>)

        a[0]=0; w[0]=0.0; sum=0.0;
        for (i=0; i<=NT; i++) sum += w[ a[i] ];
        if (sum == 0.0) sum = 1.0;
return (sum);
}
```

This function is used to calculate the job service rate. First we put the weights and the CPU queue indices into arrays, and then we calculate the sum of the weights for the terminals queueing at the CPU.

```
/*------------------------------------------------ Node definitions -------*/
         NODE cpu[cpu_states] = up;
         NODE cpuque[NT]  = 0;
         NODE repairque[NT1] = 0;
<1..NT>  NODE terminal#[term_states] = busy;
<1..NT>  NODE p#[NT]      = 0;
<1..NT>  NODE r#[NT]      = 0;
```

The structure of the nodes is similar to the reliable case. Here we define the node vector r, which is used to store the indices of the failed terminals. The size of the repairque node is NT+1, for the case when all the CPU and the terminals are down.

Let us consider the transition rules in small groups:

```
/*------------------------------------------------------ Transitions -------*/
FROM cpu[up] TO repairque,cpu[down] W cpubreak;
FROM cpu[down],repairque TO cpu[up] W cpurepair;
```

These rules used to describe the CPU breakdown and repair.

```
<1..NT><NT>    IF cpu == up FROM terminal<#1>[busy] {
                            TO terminal<#1>[waiting], cpuque, p<#2>(<#1>)
                                W prgen<#1>;
                            TO terminal<#1>[failed], repairque r<#2>(<#1>)
                                W termbreak<#1>; }
```

This rule describes the terminal breakdown and job generation transition. These events valid only if the CPU is up, and the terminal is busy.

```
<1..NT><1..NT> IF p<#2>==<#1> AND cpu == up
                            FROM cpuque, p<#2>(<#1>), terminal<#1>[waiting]
                            TO terminal<#1>[busy]
                            W prrun<#1> * weight<#1> / divider()   ;
```

This rule explains the job processing, which can occur if the CPU is up, and there is a job in the queue of the CPU. Here we call the divider function, to calculate the precise transition rate.

```
<1..NT>   IF r1==#  AND cpu == up FROM repairque, r1(#), terminal#[failed]
                                  TO terminal#[busy] W termrepair# ;
```

This is the rule of terminal repair, according to FIFO discipline among the failed terminals.

```
<2..NT>   IF p<#-1>==0    FROM p#(p#)   TO p<#-1>(p#);
<2..NT>   IF r<#-1>==0    FROM r#(r#)   TO r<#-1>(r#);
```

These non-timed rules are to deal with the "zero skipping", as introduced in 2.2.2.

```
<2..NT>   IF p<#-1> > p#  FROM p<#-1>(p<#-1>), p#(p#)
                          TOM p<#-1>(p#), p#(p<#-1>);
```

This rule is a non-timed rule, to change the neigborhood elements in the CPU queue, if they are not in increasing order. The keyword "TOM" is used here, to turn off the automatic assumption checks feature of MOSEL.

```
/*------------------------------------------------------------ Results -------*/
RESULT>> if ( cpuque != 0 AND cpu == up) cpuutil += PROB;
RESULT>> cpuquelength = MEAN cpuque;
RESULT>> if ( repairque != 0 ) repairutil += PROB;
RESULT>> repairquelength = MEAN repairque;
<1..NT> RESULT>> if (terminal# != failed) goodterminals += PROB;
<1..NT> RESULT>> if (cpu == up AND terminal# == busy) termutil# += PROB;
<1..NT> RESULT if (terminal# == waiting) termqueing# += PROB;
<1..NT> RESULT>> resptime# = termqueing# / ( prgen# * termutil# );
```

The result part is similar to the reliable one, only minor changes were neccessary.

3.3 Numerical results.

The results discussed in this section were introduced in [ALMASI99A], where the authors proved by numerical examples, that the utilization of the CPU depends on the service discipline (in the case of homogeneous sources), contrary to the reliable systems (see [ASZTALOS80], [KAMEDA82]). In the following table we can see, that the MOSEL implementation confirms the results of [ALMASI99A], by producing the same results.

Input parameters

$n = 4$	$\alpha = 0.001$	$\beta = 999.0$

i	λ_i	μ_i	γ_i	τ_i	w_i
1	0.3500	0.4000	0.2000	0.3000	3.0
2	0.3500	0.8500	0.2000	0.3000	90.0
3	0.3500	0.5000	0.2000	0.3000	15.0
4	0.3500	0.9000	0.2000	0.3000	190.0

Performance measures

	FIFO	PS	POLLING	PPS
n_j	1.283976	1.230658	1.285014	1.137364
U_r	0.754239	0.767961	0.754007	0.791032
U_{CPU}	0.663056	0.660519	0.663111	0.655889
U_1	0.268280	0.249977	0.268550	0.211706
U_2	0.292608	0.313970	0.292608	0.345383
U_3	0.276354	0.269498	0.276542	0.268845
U_4	0.294113	0.318494	0.293307	0.360611
T_1	3.785136	4.397992	3.776645	6.074148
T_2	2.909238	2.322270	2.912437	1.566298
T_3	3.475490	3.650338	3.469850	3.563455
T_4	2.860429	2.210037	2.882578	1.288600

Table 4.

A Appendix.

Reliable system with 3 terminals using FIFO discipline (long version).

```
/*
*/
/*==== Non-homogeneous terminal system input parameters =====*/
#define NT 3

<1..3> VAR double prgen# ;
<1..3> VAR double prrun# ;
/*================= The states ======================*/
enum  term_states { busy, waiting };
enum  que_states {
          S_0,          S_1,          S_2,
          S_3,          S_12,          S_13,
          S_21,          S_23,          S_31,
          S_32,          S_123,          S_132,
          S_213,          S_231,          S_312,
          S_321     };

/*================== The nodes ======================*/
        NODE system[que_states]=S_0;
<1..3>  NODE terminal#[term_states]=busy;
        NODE cpuque[3]=0;

/*=================== Transition rules ================*/
IF  system == S_0   {
<1,2,3> FROM  terminal#[busy],S_0 WITH prgen#
      TO  cpuque,terminal#[waiting],S_#;
 }
IF  system == S_1   {
    FROM  cpuque,terminal1[waiting],S_1 WITH prrun1
      TO  terminal1[busy],S_0;
<2,3> FROM  terminal#[busy],S_1 WITH prgen#
      TO  cpuque,terminal#[waiting],S_1#;
 }
IF  system == S_2   {
    FROM  cpuque,terminal2[waiting],S_2 WITH prrun2
      TO  terminal2[busy],S_0;
<1,3> FROM  terminal#[busy],S_2 WITH prgen#
      TO  cpuque,terminal#[waiting],S_2#;
 }
IF  system == S_3   {
    FROM  cpuque,terminal3[waiting],S_3 WITH prrun3
      TO  terminal3[busy],S_0;
<1,2> FROM  terminal#[busy],S_3 WITH prgen#
      TO  cpuque,terminal#[waiting],S_3#;
 }
IF  system == S_12   {
    FROM  cpuque,terminal1[waiting],S_12 WITH prrun1
      TO  terminal1[busy],S_2;
<3> FROM  terminal#[busy],S_12 WITH prgen#
      TO  cpuque,terminal#[waiting],S_12#;
 }
IF  system == S_13   {
    FROM  cpuque,terminal1[waiting],S_13 WITH prrun1
      TO  terminal1[busy],S_3;
```

```
<2> FROM  terminal#[busy],S_13 WITH prgen#
      TO  cpuque,terminal#[waiting],S_13#;
 }
IF  system == S_21   {
    FROM  cpuque,terminal2[waiting],S_21 WITH prrun2
      TO  terminal2[busy],S_1;
<3> FROM  terminal#[busy],S_21 WITH prgen#
      TO  cpuque,terminal#[waiting],S_21#;
 }
IF  system == S_23   {
    FROM  cpuque,terminal2[waiting],S_23 WITH prrun2
      TO  terminal2[busy],S_3;
<1> FROM  terminal#[busy],S_23 WITH prgen#
      TO  cpuque,terminal#[waiting],S_23#;
 }
IF  system == S_31   {
    FROM  cpuque,terminal3[waiting],S_31 WITH prrun3
      TO  terminal3[busy],S_1;
<2> FROM  terminal#[busy],S_31 WITH prgen#
      TO  cpuque,terminal#[waiting],S_31#;
 }
IF  system == S_32   {
    FROM  cpuque,terminal3[waiting],S_32 WITH prrun3
      TO  terminal3[busy],S_2;
<1> FROM  terminal#[busy],S_32 WITH prgen#
      TO  cpuque,terminal#[waiting],S_32#;
 }
IF  system == S_123   {
    FROM  cpuque,terminal1[waiting],S_123 WITH prrun1
      TO  terminal1[busy],S_23;
 }
IF  system == S_132   {
    FROM  cpuque,terminal1[waiting],S_132 WITH prrun1
      TO  terminal1[busy],S_32;
 }
IF  system == S_213   {
    FROM  cpuque,terminal2[waiting],S_213 WITH prrun2
      TO  terminal2[busy],S_13;
 }
IF  system == S_231   {
    FROM  cpuque,terminal2[waiting],S_231 WITH prrun2
      TO  terminal2[busy],S_31;
 }
IF  system == S_312   {
    FROM  cpuque,terminal3[waiting],S_312 WITH prrun3
      TO  terminal3[busy],S_12;
 }
IF  system == S_321   {
    FROM  cpuque,terminal3[waiting],S_321 WITH prrun3
      TO  terminal3[busy],S_21;
 }
/*=================== Transition rules ended ==============*/

/*======================== Results ======================*/
RESULT if ( cpuque != 0 ) cpuutil += PROB;

RESULT cpuquelength = MEAN cpuque;
<1..3> RESULT if (terminal# == busy) termutil# += PROB;
```

```
<1..3> RESULT if (terminal# == waiting) termqueing# += PROB;
/*==================== Results to print ======================*/
RESULT>> cpuutil;
RESULT>> cpuquelength;
<1..3> RESULT>> termutil#;
<1..3> RESULT>> resptime# = termqueing# / ( prgen# * termutil# );

/*
*/
```

B Appendix.

Reliable system with 3 terminals using FIFO discipline (short version).

```
/*
fifo3.msl begins */

/*=== Reliable terminal-system FIFO service for NT pieces of terminals ====*/

/*------------------------------------------------ Definitions -----------*/
#define  NT   3 // change it to 4, 5, 6, 7, ...

/*====================== No changes required below ======================*/

<1..NT> VAR double prgen# ;
<1..NT> VAR double prrun# ;

/*------------------------------------------------ Node definitions -------*/
<1..NT>  NODE terminal#[1] = 1;
          NODE cpuque[NT]  = 0;
<1..NT>  NODE p#[NT]      = 0;

/*------------------------------------------------------ Transitions -------*/
<1..NT><NT>                 FROM terminal<#1>   TO cpuque, p<#2>(<#1>)
                                W prgen<#1>;
<1..NT>       IF p1==#      FROM cpuque, p1(#)  TO terminal#
                                W prrun#   ;
<2..NT>       IF p<#-1>==0  FROM p#(p#)         TO p<#-1>(p#);

/*------------------------------------------------------------ Results -------*/
        RESULT>> cpuutil        = UTIL cpuque                        ;
        RESULT>> cpuquelength   = MEAN cpuque                        ;
<1..NT> RESULT>> termutil#      = UTIL terminal#                     ;
<1..NT> RESULT   termqueueing#  = 1-termutil#                        ;
<1..NT> RESULT>> resptime#      = termqueueing# / (prgen# * termutil#);

/* fifo3.msl ended

*/
```

C Appendix.

Reliable system with 3 terminals using PPS discipline (short version).

```
/*
pps3.msl begins */
/*=== Reliable terminal-system PPS service for NT pieces of terminals ====*/

/*------------------------------------------------ Macro definition -------*/
#define  NT   3 // change it to 4, 5, 6, 7, ... MAXNT

/*====================== No changes required below ======================*/
#string  w_assign  w[#]=weight#; :
#string  a_assign  a[#]=p#;      :

#define  MAXNT 15

<1..NT> VAR double prgen# ;
<1..NT> VAR double prrun# ;
<1..NT> VAR double weight# ;
/*------------------------------------------- Function definitions -------*/
double divider(void)
{  int i,a[MAXNT+1];
   double sum, w[MAXNT+1];

/*======== Put the weights and the cpu queue indices into arrays =========*/
        $w_assign(<1..NT>)
        $a_assign(<1..NT>)

a[0]=0; w[0]=0.0; sum=0.0;
for (i=0; i<=NT; i++) sum += w[ a[i] ];
if (sum == 0.0) sum = 1.0;
return (sum);
}
/*------------------------------------------------ Node definitions -------*/
<1..NT>  NODE terminal#[1] = 1;
         NODE cpuque[NT]  = 0;
<1..NT>  NODE p#[NT]      = 0;
/*---------------------------------------------------- Transitions -------*/
<1..NT><NT>                   FROM terminal<#1>   TO cpuque, p<#2>(<#1>)
W prgen<#1>;
<1..NT><1..NT> IF p<#2>==<#1> FROM cpuque, p<#2>(<#1>)  TO terminal<#1>
W prrun<#1> * weight<#1> / divider()   ;

<2..NT>        IF p<#-1>==0  FROM p#(p#)         TO p<#-1>(p#);

<2..NT> IF p<#-1> > p#
FROM p<#-1>(p<#-1>), p#(p#)  TOM p<#-1>(p#), p#(p<#-1>);

/*------------------------------------------------------------ Results -------*/
         RESULT>> cpuutil        = UTIL cpuque                        ;
         RESULT>> cpuquelength   = MEAN cpuque                        ;
<1..NT> RESULT>> termutil#      = UTIL terminal#                     ;
<1..NT> RESULT   termqueueing#  = 1-termutil#                        ;
<1..NT> RESULT>> resptime#      = termqueueing# / (prgen# * termutil#);
/* pps3.msl ended
*/
```

D Appendix.

Reliable system with 3 terminals using POLLING discipline (short version).

```
/*
poll3.msl begins */

/*== Reliable terminal-system POLLING service for NT pieces of terminals ==*/

/*------------------------------------------------ Definitions ------------*/
#define  NT   3 // change it to 3, 4, 5, 6, 7, ...

/*====================== No changes required below ======================*/

<1..NT> VAR double prgen# ;
<1..NT> VAR double prrun# ;
/*------------------------------------------------ Node definitions -------*/
<1..NT>  NODE terminal#[1] = 1;
         NODE cpuque[NT]   = 0;
<1..NT>  NODE p#[NT]       = 0;


/*------------------------------------------------------ Transitions -------*/
<1..NT><NT>            FROM terminal<#1>   TO cpuque, p<#2>(<#1>)
                              W prgen<#1> ;
<1..NT>  IF p1==#     FROM cpuque, p1(#)  TO terminal#
                              W prrun# ;

<2..NT>  IF p<#-1>==0 FROM p#(p#) TO p<#-1>(p#);
<3..NT> IF (0 < p<#-1> AND p<#-1> < p1 AND p1 < p# ) OR
           (0 < p# AND p# < p<#-1> AND p<#-1> < p1 ) OR
           (0 < p1 AND p1 < p# AND p# < p<#-1> )
                                        FROM p<#-1>(p<#-1>), p#(p#)
                                         TOM p<#-1>(p#),     p#(p<#-1>);

/*------------------------------------------------------------ Results -------*/
        RESULT>> cpuutil        = UTIL cpuque                        ;
        RESULT>> cpuquelength   = MEAN cpuque                        ;
<1..NT> RESULT>> termutil#      = UTIL terminal#                     ;
<1..NT> RESULT   termqueueing#  = 1-termutil#                        ;
<1..NT> RESULT>> resptime#      = termqueueing# / (prgen# * termutil#);

/* poll3.msl ended

*/
```

E Appendix.

Non-reliable system with 3 terminals using FIFO discipline (short version).

```
/*
fifonr3.msl begins */
/*== Non-eliable terminal-system FIFO service for NT pieces of terminals ==*/

/*------------------------------------------------ Macro definition -------*/
#define  NT   3 // change it to 4, 5, 6, 7, ... MAXNT

/*====================== No changes required below ======================*/
<NT> #define  NT1 <#+1>
        VAR double cpubreak;
        VAR double cpurepair;
<1..NT> VAR double termbreak# ;
<1..NT> VAR double termrepair# ;
<1..NT> VAR double prgen# ;
<1..NT> VAR double prrun# ;

enum term_states {busy, waiting, failed};
enum cpu_states {up, down};
/*------------------------------------------------ Node definitions -------*/
         NODE cpu[cpu_states] = up;
         NODE cpuque[NT]  = 0;
         NODE repairque[NT1] = 0;
<1..NT>  NODE terminal#[term_states] = busy;
<1..NT>  NODE p#[NT]      = 0;
<1..NT>  NODE r#[NT]      = 0;
/*------------------------------------------------------ Transitions -------*/
FROM cpu[up] TO repairque,cpu[down] W cpubreak;
FROM cpu[down],repairque TO cpu[up] W cpurepair;

<1..NT><NT>   IF cpu == up  FROM terminal<#1>[busy] {
                            TO terminal<#1>[waiting], cpuque, p<#2>(<#1>)
                                W prgen<#1>;
                            TO terminal<#1>[failed], repairque r<#2>(<#1>)
                                W termbreak<#1>; }

<1..NT>   IF p1==#  AND cpu==up  FROM cpuque, p1(#), terminal#[waiting]
                                     TO terminal#[busy] W prrun# ;
<1..NT>   IF r1==#  AND cpu==up  FROM repairque, r1(#), terminal#[failed]
                                     TO terminal#[busy] W termrepair# ;

<2..NT>       IF p<#-1>==0  FROM p#(p#)         TO p<#-1>(p#);
<2..NT>       IF r<#-1>==0  FROM r#(r#)         TO r<#-1>(r#);
/*------------------------------------------------------ Results -------*/
RESULT>> if ( cpuque != 0 AND cpu == up) cpuutil += PROB;
RESULT>> cpuquelength = MEAN cpuque;
RESULT>> if ( repairque != 0 ) repairutil += PROB;
RESULT>> repairquelength = MEAN repairque;
<1..NT> RESULT>> if (terminal# != failed) goodterminals += PROB;
<1..NT> RESULT>> if (cpu == up AND terminal# == busy) termutil# += PROB;
<1..NT> RESULT if (terminal# == waiting) termqueing# += PROB;
<1..NT> RESULT>> resptime# = termqueing# / ( prgen# * termutil# );
/* fifonr3.msl ended

*/
```

F Appendix.

Non-reliable system with 3 terminals using PPS discipline (short version).

```
/*
ppsnr3.msl begins */

/*== Non-reliable terminal-system PPS service for NT pieces of terminals ===*/

/*------------------------------------------------ Macro definition -------*/
#define  NT   3 // change it to 4, 5, 6, 7, ... MAXNT

/*====================== No changes required below ======================*/
#string  w_assign  w[#]=weight#; :
#string  a_assign  a[#]=p#;      :

#define  MAXNT 15

<NT>    #define NT1 <#+1>

        VAR double cpubreak;
        VAR double cpurepair;
<1..NT> VAR double termbreak# ;
<1..NT> VAR double termrepair# ;
<1..NT> VAR double prgen# ;
<1..NT> VAR double prrun# ;
<1..NT> VAR double weight# ;

enum term_states {busy, waiting, failed};
enum cpu_states {up, down};
/*------------------------------------------- Function definitions -------*/
double divider(void)
{  int i,a[MAXNT+1];
   double sum, w[MAXNT+1];

/*======== Put the weights and the cpu queue indices into arrays =========*/
        $w_assign(<1..NT>)
        $a_assign(<1..NT>)

a[0]=0; w[0]=0.0; sum=0.0;
for (i=0; i<=NT; i++) sum += w[ a[i] ];
if (sum == 0.0) sum = 1.0;
return (sum);
}
/*------------------------------------------------ Node definitions -------*/
         NODE cpu[cpu_states] = up;
         NODE cpuque[NT]  = 0;
         NODE repairque[NT1] = 0;
<1..NT>  NODE terminal#[term_states] = busy;
<1..NT>  NODE p#[NT]      = 0;
<1..NT>  NODE r#[NT]      = 0;
/*------------------------------------------------- Transitions -------*/
FROM cpu[up] TO repairque,cpu[down] W cpubreak;
FROM cpu[down],repairque TO cpu[up] W cpurepair;

<1..NT><NT>    IF cpu == up FROM terminal<#1>[busy] {
                            TO terminal<#1>[waiting], cpuque, p<#2>(<#1>)
                                W prgen<#1>;
```

```
                            TO terminal<#1>[failed], repairque r<#2>(<#1>)
                                W termbreak<#1>; }

<1..NT><1..NT> IF p<#2>==<#1> AND cpu == up
                            FROM cpuque, p<#2>(<#1>), terminal<#1>[waiting]
                            TO terminal<#1>[busy]
                            W prrun<#1> * weight<#1> / divider()   ;
<1..NT>   IF r1==#  AND cpu == up FROM repairque, r1(#), terminal#[failed]
                                  TO terminal#[busy] W termrepair# ;

<2..NT>   IF p<#-1>==0    FROM p#(p#)   TO p<#-1>(p#);
<2..NT>   IF r<#-1>==0    FROM r#(r#)   TO r<#-1>(r#);
<2..NT>   IF p<#-1> > p#  FROM p<#-1>(p<#-1>), p#(p#)
                          TOM p<#-1>(p#), p#(p<#-1>);

/*------------------------------------------------------------ Results -------*/
RESULT>> if ( cpuque != 0 AND cpu == up) cpuutil += PROB;
RESULT>> cpuquelength = MEAN cpuque;
RESULT>> if ( repairque != 0 ) repairutil += PROB;
RESULT>> repairquelength = MEAN repairque;
<1..NT> RESULT>> if (terminal# != failed) goodterminals += PROB;
<1..NT> RESULT>> if (cpu == up AND terminal# == busy) termutil# += PROB;
<1..NT> RESULT if (terminal# == waiting) termqueing# += PROB;
<1..NT> RESULT>> resptime# = termqueing# / ( prgen# * termutil# );

/* ppsnr3.msl ended

*/
```

G Appendix.

Non-reliable system with 3 terminals using POLLING discipline (short version).

```
/*
pollnr3.msl begins */

/* Non-reliable terminal-system POLLING service for NT pieces of terminals */

/*------------------------------------------------ Macro definition -------*/
#define  NT   3 // change it to 4, 5, 6, 7, ...

/*====================== No changes required below ======================*/
<NT> #define  NT1 <#+1>

        VAR double cpubreak;
        VAR double cpurepair;
<1..NT> VAR double termbreak# ;
<1..NT> VAR double termrepair# ;
<1..NT> VAR double prgen# ;
<1..NT> VAR double prrun# ;


enum term_states {busy, waiting, failed};
enum cpu_states {up, down};
/*------------------------------------------------ Node definitions -------*/
         NODE cpu[cpu_states] = up;
         NODE cpuque[NT]  = 0;
         NODE repairque[NT1] = 0;
<1..NT>  NODE terminal#[term_states] = busy;
<1..NT>  NODE p#[NT]      = 0;
<1..NT>  NODE r#[NT]      = 0;

/*------------------------------------------------------ Transitions -------*/
FROM cpu[up] TO repairque,cpu[down] W cpubreak;
FROM cpu[down],repairque TO cpu[up] W cpurepair;

<1..NT><NT>   IF cpu == up  FROM terminal<#1>[busy] {
                            TO terminal<#1>[waiting], cpuque, p<#2>(<#1>)
                                W prgen<#1>;
                            TO terminal<#1>[failed], repairque r<#2>(<#1>)
                                W termbreak<#1>; }


<1..NT>   IF p1==#  AND cpu==up  FROM cpuque, p1(#), terminal#[waiting]
                                     TO terminal#[busy] W prrun# ;
<1..NT>   IF r1==#  AND cpu==up  FROM repairque, r1(#), terminal#[failed]
                                     TO terminal#[busy] W termrepair# ;

<2..NT>       IF p<#-1>==0  FROM p#(p#)         TO p<#-1>(p#);
<2..NT>       IF r<#-1>==0  FROM r#(r#)         TO r<#-1>(r#);

<3..NT> IF (0 < p<#-1> AND p<#-1> < p1 AND p1 < p# ) OR
           (0 < p# AND p# < p<#-1> AND p<#-1> < p1 ) OR
           (0 < p1 AND p1 < p# AND p# < p<#-1> )
                                          FROM p<#-1>(p<#-1>), p#(p#)
                                           TOM p<#-1>(p#),     p#(p<#-1>);
```

```
/*------------------------------------------------------- Results -------*/
RESULT if ( cpuque != 0 AND cpu == up) cpuutil += PROB;
RESULT if ( repairque != 0 ) repairutil += PROB;
RESULT repairquelength = MEAN repairque;
RESULT cpuquelength = MEAN cpuque;
<1..NT> RESULT if (cpu == up AND terminal# == busy) termutil# += PROB;
<1..NT> RESULT if (terminal# == waiting) termqueing# += PROB;
<1..NT> RESULT if (terminal# != failed) goodterminals += PROB;
/*====================== Results to print ======================*/
RESULT>> cpuutil;
RESULT>> cpuquelength;
RESULT>> repairutil;
RESULT>> repairquelength;
RESULT>> goodterminals;
<1..NT> RESULT>> termutil#;
<1..NT> RESULT>> resptime# = termqueing# / ( prgen# * termutil# );

/* pollnr3.msl ended

*/
```

H Appendix.

The C programs to generate the long MOSEL program showed in Appendix A

```
/*
term.c begins */
/************************************************************************

 term.c version 2.0; University Erlangen, Erlangen, 1999.

 Author: Almasi, Bela University Debrecen, Hungary, almasi@math.klte.hu

 Generates mosel input file (NODE-s, RULE-s, RESULT-s) for modeling
 non-homogeneous terminal systems. Concerning the service disciplines
 see below.

 Conventions, notations(in this program, and in the MOSEL file):
 ==============================================================
 NT denotes the number of terminals (default value is 3).
 prgeni (i=1,2,...,NT) denotes the program generation intensity for
                    terminal i.
 prruni (i=1,2,...,NT) denotes the program running intensity for
                    terminal i.
 S_ijk denotes the que, where jobs from terminal i,j,k are in the que.

 cpuutil denotes the utilization of the cpu.
 quelength denotes the length of the que.
 termutili (i=1,..,NT) denotes the utilization of terminal i.
 termqueingi (i=1,...,NT) denotes the probability that the job of terminal i
                    is in the que.
 resptimei (i=1,...NT) denotes the response time for terminal i.


 Command line parameter: NT (default value: 3)
 Output: The complete MOSEL input file is written to
          the standard output.

 Usage: term [No_of_terminals]


To compile, the program needs the following functions (from an other file):
============================================================================
void deletejob(char *old, char job, char *new)
     Creates the new statevector from old by servicing the given job.
void addjob(char *old, char job, char *new)
     Creates the new statevector from old by adding the given new job.
void nextstate(char *state )
     Calculates the following state. state is I/O.
     if state[NT-1] == NT+1 at the returned state, it means no more states.
void generatejob(char *state)
     Prints the MOSEL rule WITH and TO parts for generating job(s) starting
     from the given state.
void servicejob(char *state)
     Prints the MOSEL rule WITH and TO parts for servicing job(s) starting
     from the given state.
char useweights()
     Returns 1, if the actual service discipline uses priorities (weights),
             0, otherwise.
```

```
These functions depend on the service discipline, and must be defined
in a file separately for the different service disciplines.
This file (term.c) is common for all service disciplines.
Compilation(for example for fifo rule): gcc -o term fifo.c term.c

The header file term.h contains something like this:
#define MAXNT 9
#define lambda "prgen"
#define mue "prrun"
#define weight "weight"
char NT;

CHANGES
=======

v2.0 Introducing new nodes (terminal, que), so the RESULT part will
     be much more shorter. Running time of mosel will be 30% less.

*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "term.h"


/***************** Calculating the length of the state vector ********/
char statelength(char *state)
{ char i;
i=0;
while (state[i]!=0) i++;
return(i);
}


/****************** Print the state vector of length NT ***********/
void printstate(char *state)
{ char i;
printf("S_");
if (state[0] == 0 ) printf("0");
else
   for (i=NT-1; i>=0; i--)
if (state[i] == '#' ) printf("#");
    else if (state[i] > 0) printf("%1d",state[i]);
}


/****************** Print the shortcut from vector state ***********/
void printshortcut(char *state)
{ char i;
printf("<%1d",state[0]);
for (i=1; i<statelength(state); i++) printf(",%1d",state[i]);
printf(">");
}


/********* Testing if state contains c from place i up to NT ***/
char contains(char *state, char i, char c)
```

```
{ char j;
for (j=i; j<NT; j++)
    if (state[j] == c ) return(1);
return(0);
}


/********** Generating the input parameters, and state space **************/
void paramgen ()
{
char i,sl;
char c,sst[MAXNT+1],state[MAXNT+1];

printf("/*\n%cbegin{verbatim}\n*/\n",'\\'); /* for tex input */
printf("/*==== Non-homogeneous terminal system input parameters =====*/\n");

printf("#define NT %d\n\n",NT);
printf("<1..NT> VAR double prgen# ;\n");
printf("<1..NT> VAR double prrun# ;\n");

if (useweights()) for (i=1;i<=NT;i++) printf("#define %s%d 1.0\n",weight,i);

printf("/*================= The states ========================*/\n");
printf("enum  term_states { busy, waiting };\n");
printf("enum  que_states {  \n");

for (i=0; i <= MAXNT+1; i++) state[i] = 0;
i=0;
while ( state[NT-1] != NT+1 ) /*** if NT+1 reached, we are ready */
   { i++;
        printf("          ");printstate(state);
nextstate(state);
        if (state[NT-1] != NT+1) printf(",");
        if ( i == 3) {i=0; printf("\n");}
   }
printf("     };\n\n");
}


/********************** Generating the nodes **********************/
void nodegen ()
{
char i;
char state[MAXNT+1];
for (i=0; i <= MAXNT+1; i++) state[i] = 0;
printf("/*================== The nodes ====================*/\n");
printf("        NODE system[que_states]=");printstate(state);printf(";\n");
printf("<1..%d>  NODE terminal#[term_states]=busy;\n",NT);
printf("        NODE cpuque[%d]=0;\n",NT);
printf("\n");
}


/***************** Generating the transition rules  *****************/
void rulegen ()
{
char i,sl;
char c,sst[MAXNT+1],state[MAXNT+1];
```

```
printf("/*================== Transition rules =================*/\n");
for (i=0; i <= MAXNT+1; i++) state[i] = 0;
while ( state[NT-1] != NT+1 ) /*** if NT+1 reached, we are ready */
   { sl=statelength(state);
printf("IF  system == ");printstate(state); printf("   {\n");
        if (sl > 0) servicejob(state);  /** service is possible***/
if (sl< NT) generatejob(state); /** new job generation is possible */
printf(" }\n");
nextstate(state);
    }
printf("/*================== Transition rules ended ==============*/\n\n");
}


/************* Generating the results ****************************/
void resultgen ()
{
char i,sl,si;
char c,sst[MAXNT+1],state[MAXNT+1],shortcut[MAXNT+1];

printf("/*========================= Results ======================*/\n");
for (i=0; i <= MAXNT+1; i++) state[i] = 0;

printf("RESULT if ( cpuque != 0 ) cpuutil += PROB;\n\n");
printf("RESULT cpuquelength = MEAN cpuque;\n");
printf("<1..%d> RESULT if (terminal# == busy) termutil# += PROB;\n",NT);
printf("<1..%d> RESULT if (terminal# == waiting) termqueing# += PROB;\n",NT);

/****** Noe we have the results, let us print them ***********************/
printf("/*====================== Results to print ======================*/\n");
printf("RESULT>> cpuutil;\n");
printf("RESULT>> cpuquelength;\n");
printf("<1..%d> RESULT>> termutil#;\n",NT);
/*printf("<1..%d> RESULT>> termqueing#;\n",NT);*/
printf("<1..%d> RESULT>> resptime# = termqueing# / ( %s# * termutil# );\n",
       NT, lambda);
printf("\n");
printf("/*\n%cend{verbatim}\n*/\n",'\\'); /* for tex input */
}


/************************* Main program ********************************/
void main(int argc,char **argv)
{
NT = 3;                                 /* default value NT=3 */
if (argc == 2) NT = atoi(argv[1]);    /* or the parameter given */

paramgen();
nodegen();
rulegen();
resultgen();
}

/* term.c ended

*/
```

```
/*
fifo.c begins */
/*************************************************************************

 fifo.c version 2.0; University Erlangen, Erlangen, 1999.

 Author: Almasi, Bela University Debrecen, Hungary, almasi@math.klte.hu

 The required functions for term.c v. 2.0 using fifo service discipline.

*************************************************************************/

#include "term.h"

/************* Indicating that FIFO does not use priorities ***************/
char useweights()
{
return(0);
}



/******* Create the new statevector from old by servicing a job *******/
void deletejob(char *old, char job, char *new)
{ char i,c;

c=0; new[NT-1]=0;
for (i=0; i<NT-1; i++)
    {
if (old[i] == job) c=1;
new[i]=old[i+c];
    }
}



/******* Create the new statevector from old by adding a new job *******/
void addjob(char *old, char job, char *new)
{
char i,c;

new[0]=job;
for (i=0; i<NT-1; i++) new[i+1]=old[i];
}


/* Prints the WITH and TO part of rules servicing jobs from the given state */
void servicejob(char *state)
{
char c, sst[MAXNT+1];

c=state[statelength(state)-1];
deletejob(state,c,sst);
printf("    FROM  cpuque,terminal%d[waiting],",c);printstate(state);
printf(" WITH %s%1d \n",mue,c,c);
printf("      TO  terminal%d[busy],",c);printstate(sst);
printf(";\n");
}
```

```
/*** Prints the WITH and TO part of rules creating jobs to the given state **/
void generatejob(char *state)
{
char c, sst[MAXNT+1], shortcut[MAXNT+1];

addjob(state,'#',sst);
for (c=0; c<=NT; c++) shortcut[c]=0;
for (c=1; c<=NT; c++)
    if (!contains(state,0,c)) shortcut[ statelength(shortcut) ] = c;

printshortcut(shortcut);
printf(" FROM  terminal#[busy],"); printstate(state);
printf(" WITH %s# \n",lambda);
printf("      TO  cpuque,terminal#[waiting],");
printstate(sst);
printf(";\n");

}



/********* Calculating the following state (the next one)********/
void nextstate(char *state )
{
char i,j;
char c, ok;

ok = 0;
for (i=0; (i<NT) && (!ok); i++ )
    {   c = state[i]+1;
while ( (c <= NT) && (!ok) )
     if (!contains(state,i,c))
  {ok=1; state[i]=c;}
     else c++;
        state[i]=0;
    }
/***  c=NT+1 can happen, that means, the input state was the last one ***/

i--;
state[i]=c;
for (j=i-1; j>=0; j--)
{   ok=0;
    for (c=1; (c<=NT) && (!ok); c++)
        if (!contains(state,j,c)) {ok=1; state[j]=c;}
}

}

/* fifo.c ended

*/
```

References

[ALMASI92] Almási, B: A Queuing Model for a Processor-Shared Multi-Terminal System Subject to Breakdowns, *Acta Cybernetica* Vol. 10, Nr. 4, 273-282, (1996).

[ALMASI96] Almási, B: Response Time for Finite Heterogeneous Nonreliable Queueing Systems, *Computers and Mathematics with Applications* Vol. 31, No. 11, 55-59, (1996).

[ALMASI99] Almási, B.: A Queueing Model for a Non-homogeneous Polling System Subject to Breakdowns *Annales Univ. Sci. Budapest. Sec. Comp.* (to appear 1999).

[ALMASI99A] Almási, B. and Sztrik, J.: Optimization problems on the performance of a non-reliable terminal system, *Computers and Mathematics with Applications* Vol. 38, No. 3, (1999 to appear).

[BOLCH99] Bolch, G. and Herold, H.: MOSEL MOdeling Specifocation and Evaluation Language *Technical Report,* University Erlangen (1999).

[CIARDO91] Ciardo, G.; Muppala, J.K.: Manual for the SPNP Package Version 3.1, Duke University, Durham, NC, USA, 1991.

[CSIGE82] Csige, L. and Tomkó, J.: The machine interference for exponentially distributed operating and repair times *Alk. Mat. Lapok* 8, 107–124, (1982), (in Hungarian).

[SZTRIK86] Sztrik, J.: A probability model for priority processor-shared multiprogrammed computer systems, *Acta Cybernetica* 7, 329–340, (1986).

[SZTRIK90] Sztrik, J. and Gál, T.: A recursive solution of a queueing model for a multi-terminal system subject to breakdowns. *Performance Evaluation* 11, 1–7, (1990).

[TRIVEDI91] Trivedi, K.S.; Ciardo, G.: A Decomposition Approach for Stochastic Reward Net Models, Duke University, Durham, NC, USA, (1991).

[ASZTALOS80] Asztalos, D. Optimal control of finite-source priority queues with computer system applications, *Computers and Mathematics with Applications* 6, 425-431, (1980).

[KAMEDA82] Kameda, H. A finite-source queue with different customers. J. ACM 29, 478-491, (1982).