Generation and Solution of Markov Chains Using MOSES

Gunter Bolch and Stefan Greiner

May 1994 TR-I4-11-94

Generation and Solution of Markov Chains Using MOSES

Gunter Bolch Stefan Greiner Hermann Jung Raimund Zimmer

University Erlangen-Nuernberg
Institute for mathematical machines and dataprocessing IV
Martensstrasse 1
D – 91058 Erlangen

bolch@informatik.uni-erlangen.de

Technical Report TR-I4-11-94

Abstract

In this paper the Markov analyzer **MOSES** (**MO**delling, **S**pecification and **E**valuation **S**ystem) and the model description language **MOSLANG** – both developed at the Institute for Operating Systems at the University of Erlangen–Nuernberg – are described using two examples. To evaluate the performance of a computer system the system has to be specified. In this paper the model description language **MOSLANG** is introduced and applied to some examples. The core of MOSLANG consists of constructs suitable for the specification of the possible states of the system and of RULE constructs which model the state transitions. This specification method is much more compact than other comparable methods and enables the user to specify large systems when he has become familiar with MOSLANG. The Markov analyzer **MOSES** supports the input of MOSLANG and subsequently creates the Markovian system of equations automatically. For solving this system of equations five different methods are provided. MOSES calculates the state probabilities and derives from them the performance measures which are specified using MOSLANG. MOSES has been used to solve many different problems, among others for analyzing and comparing different multiprocessor versions of the operating system UNIX. MOSES is implemented in the programming language C and runs on all UNIX systems.

1 Introduction

In todays systems performance evaluation is very important because of the growing system complexity and should therefore go hand by hand with the developement of the real system. Therefore measurement on the real system is very often not possible and one needs a model of the system. In this paper we will concentrate on queueing models but the description language MOSLANG is not only restricted to this type of model.

Queueing models are very well suited for the performance evaluation of computer systems because they are easy to understand and describe the system in a very compact manner. For the limited class of productform queueing networks (PFQN's) there exist efficient algorithms (e.g. Mean Value Analysis, SCAT, Convolution) to determine the desired performance measures of the system. But a lots of queueing networks do not fulfil the requirements of PFQN's. These networks are called non productform queueing networks (NPFQN's) and are analyzed by approximate algorithms. If these algorithms are not exact enough or not applicable to a particular problem at all – this can be the case when rates are non exponentially distributed or blocking is involved – then one normally uses simulation with its well known disadvantages or Markovian analysis.

Although the state space for simple systems can be very large and grows exponentially in the system complexity, Markov analysis becomes more and more important because new mathematical methods for

solving large equation systems and increasing computational power enables scientists to get steady state and transient performance measures in an acceptable time compared to simulation. The problem with this models is that it is nearly impossible to construct the Markov model for a complex system by hand. One possibility for solving this problem is by modeling the system via stochastic Petri nets. These offer a much higher modeling power than queueing networks. A very famous tool for dealing with Petri nets is SPNP (developed at Duke University) which converts the specification of a Petri net into a Markov chain and solves the underlying equation system.
In this paper another method is choosen to specify a system. With the help of a new language called **MOSLANG** the state space of the Markov chain can be described in a compact manner. The core of MOSLANG consists of constructs which allow the specification of the possible states of the system and the transitions between the states. This specification method allowes a very compact specification of the system compared to other methods and enables the experienced user to specify large systems very fast.
The Markov analyzer **MOSES** facilitates the input of MOSLANG and creates the Markovian system of equations $\underline{p} \cdot \underline{Q} = 0$ automatically. Here $\underline{p}$ is the vector of the state probabilities and $\underline{Q}$ is the matrix which contains the transition rates between the different states of the Markov chain. $\underline{Q}$ is created in a parameterized form. Therefore the system can be analyzed with different parameters without changing the Markovian system each time. Solving this system provides the state probabilities. For this purpose five methodes are supported. With the state probabilities it is possible to compute all specified measures automatically using MOSES.
In the next two sections we will describe the main charactristics of MOSLANG and show how to use this tool by specifying two systems – a simple fault tolerant multiprocessor system and the model of a fault tolerant UNIX operating system.

2 Generation and solution of Markov chains

With the specification language **MOSLANG** it is possible to describe the Markovian system in a very compact manner. Out of this description the state space ist generated and the underlying set of equations is solved. This is done in several steps.

2.1 General description

With the help of the description language **MOSLANG** the system is described. Out of this description the state diagram is generated. This state diagram consists of *stable* and *unstable* satates. A state is called unstable if the Markov process does not stay in that state but changes immediately in another state. Immediate transitions are indicated by the rate -1.0. To be able to generate a state transition matrix it is necessary to transform the state diagram into an equivalent state diagram which only consists of positive rates. Out of this compressed state diagram we generate the state transition matrix and compute the state probabilities.
Therefore the Markov analyzer MOSES performs the following steps when solving the system :

1. generation of the state diagram.
2. compression of the state diagram.
3. generation of the state transition matrix.
4. computation of the state probabilities.

Of special interest here are the first two steps because it is very time consuming to generate the state diagram and then to compress this state diagram. In our tool we use a technique where the compressed state diagram is generated without generating the "standard" state diagram.
In the following description it is assumed that timeless transitions have higher priority than transitions with positive transition rates.

2.2 Compression of the state diagram

Notation 1 *For an element (A,B,r) of a state diagram we write*

$$A \xrightarrow{\mathrm{r}} B$$

to indicate that there is a transition of state A to state B with rate r.

Notation 2 *For an arc $k = A \xrightarrow{r} B$ of a graph , it is*

$$\begin{aligned} s(k) &\stackrel{def}{=} A \\ z(k) &\stackrel{def}{=} B \\ r(k) &\stackrel{def}{=} r \end{aligned}$$

Definition 1 *A state diagram , is a finite set of triples (A, B, r), $r \neq 0$. Each triple represents a state transition of the system from state A to state B with rate r.*

Definition 2 *In the state diagram , the set $Z[,]$ is defined as*

$$Z[,] \stackrel{def}{=} \{X \mid \exists Y \exists r : (X \xrightarrow{r} Y \in , \vee Y \xrightarrow{r} X \in ,)\}$$

Definition 3 *1. in a state diagram , the set of all pathes $\Psi[,]$ is defined as*

$$\Psi[,] \stackrel{def}{=} \{(k_1, k_2, \ldots, k_n) \in ,^* \mid z(k_1) = s(k_2) \wedge \ldots \wedge z(k_{n-1}) = s(k_n)\}$$

.

2. the elements of $\Psi[,]$ are called pathes.

3. for each path $p = (k_1, k_2, \ldots, k_n)$ we have

$$\begin{aligned} \mid p \mid &\stackrel{def}{=} n \\ p_i &\stackrel{def}{=} k_i \quad \forall i = 1 \ldots n. \end{aligned}$$

4. all pathes with $s(p_1) = z(p_{|p|})$ are called cycles.

Definition 4 *A graph is called standardized $\Longleftrightarrow$*

$$\forall A \in Z[,] : A \quad unstable \quad \Longrightarrow \sum_{\substack{s(k)=A, r(k)<0 \\ k \in \Gamma}} r(k) = -1$$

Definition 5 *An arc $k = A \xrightarrow{r} B \in ,$ is called essential (in short ess_Γ) $\Longleftrightarrow$*

$$\begin{aligned} & r < 0 \quad \vee \\ \forall \quad C, s : A \xrightarrow{s} C \in , \quad & \Longrightarrow \quad s > 0 \end{aligned}$$

Definition 6 *Let $A, B \in ,$ then the set $N^*(A,B)$ is defined as*

$$\begin{aligned} N^*(A, B) &\stackrel{def}{=} \{p \in \Psi[,] \mid \\ s(p_1) = A \wedge z(p_{|p|}) = B \quad &\wedge \quad r(p_1) > 0 \wedge (\forall i > 1 : r(p_i) < 0) \wedge B \text{ is stable}\} \end{aligned}$$

Definition 7 *B is called a sucessor* state of A in ,* $\quad \Longleftrightarrow \quad N^*(A, B) \neq \{\}$

At first we assume that we have still a "standard" state diagram , which is still not compressed. Then the following steps have to be carried out [MUNK91]:

Transformation 1 (Standardization of negative transitions) *The graph* $,_{norm}$ *is the standardized equivalent of the graph* , $\iff$

$$
\begin{aligned}
& ,_{norm} \text{ is standardized } \wedge \\
& \forall A \in Z[,]: \exists k > 0 : \forall B, r : \quad (A \xrightarrow{r} B \in , \iff \\
& \qquad (r > 0 \wedge A \xrightarrow{\mathrm{r}} B \in ,_{norm}) \vee \\
& \qquad (r < 0 \wedge A \xrightarrow{\mathrm{k \cdot r}} B \in ,_{norm}))
\end{aligned}
$$

Transformation 2 (Removing of positive arcs that lead into unstable states) *The graph* $,_{ess} \subseteq ,$ *is the essential equivalent of the graph* , $\iff$

$$\forall A \xrightarrow{\mathrm{r}} B \in , : ess_{\Gamma}(A \xrightarrow{\mathrm{r}} B) \iff A \xrightarrow{\mathrm{r}} B \in ,_{ess}$$

Transformation 3 (Removing of non recurrent states) *The graph* $,_{red} \subseteq ,$ *is called the reduced equivalent of the graph* , $\iff$

$$
\begin{aligned}
& \forall A \in Z[,]: \forall p \in \Psi[,]: s(p_1) = A : \\
& \qquad (\exists n \geq 1 : \exists k_1, k_2, \ldots, k_n \in , : \\
& \qquad k_1 = A \xrightarrow{\mathrm{r_1}} X_1, k_2 = X_1 \xrightarrow{\mathrm{r_2}} X_2, \ldots, k_n = X_{n-1} \xrightarrow{\mathrm{r_n}} A) \\
& \iff \forall i : p_i \in ,_{red}
\end{aligned}
$$

Transformation 4 (Remove immediate transitions) *The graph* $,'$ *is the compressed equivalent of the graph* , $\iff$

$$
\begin{aligned}
\forall A, B \in Z[,] \quad : \quad & B \text{ is the successor}^* \text{ state of } A \iff \\
& A \xrightarrow{\mathrm{r}} B \in ,' \text{ where } r = \sum_{p \in N^*(a,b)} \prod_{1 \leq i \leq |p|} | r(p_i) |
\end{aligned}
$$

The Markovian process of a graph $,'$ which is generated by carrying out the four transformation steps 1, 2, 3 and 4 has the same observable behaviour than the original graph , but the graph $,'$ consists only of positive arcs. To be able to compute the state probabilities the graph $,'$ must be irreducable. Therefore the graph $,_3$ which we get from transformation 3 must be homogenuous and must not have cycles that consist only of timeless transitions. Furthermore the reduced equivalent of $,_3$ must be irreducable. Because of the fact that the transformation steps 1, 2 and 3 do not add arcs to the system we have: if , has no cycles that consist only of timeless transitions then $,_3$ has also no cycles that consist only of timeless transitions.

Therefore for a state diagram that can be analyzed with the tool MOSES the following *consistency condition* must hold:

> The specified state diagram , must not have cycles that consist only of timeless transitions. Furthermore the reduced essential standardized equivalent of , must be irreducable.

For practical use like the modelling of queueing systems in steady state this consistency condition is normally fulfiled.

2.3 Generation of the state space

In this section it is introduced a new method [MUNK91] which generates the compressed state diagram without generating the "original" state diagram , . In this new method states and arcs are only generated if they belong to the compressed graph. The presented algorithm is based on the following observation: if one knows a state of the compressed state diagram then all other states of the compressed state diagram can be reached from this state because the consistency condition demands that the compressed state diagram is irreducable.
The algorithm presented in Fig. 1 uses as starting state any stable state of the non compressed graph. In the rest of this section it is proven that this algorithm is functional equivalent to the transformation steps 1 to 4 (for detail see [MUNK91]).

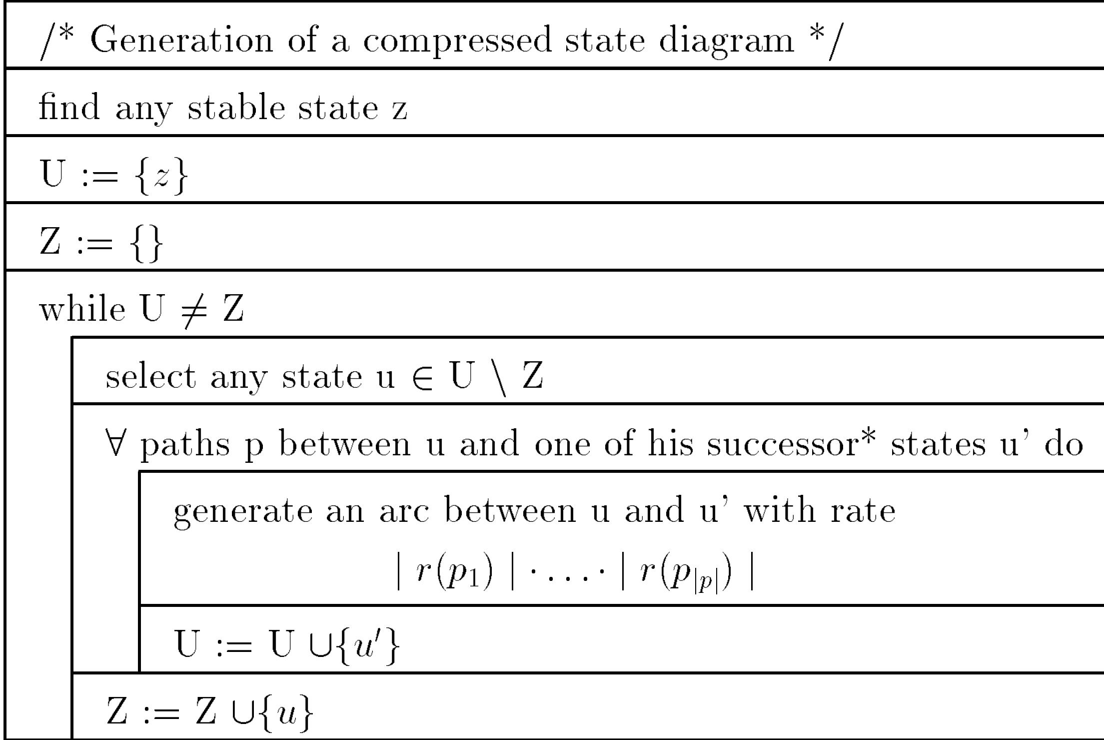

Figure 1: Generation of the compressed state diagram

For the rest of this section we call this new algorithm *transformation N*. In order to make the proof easier we presuppose that *transformation N* still works on the essential standardized equivalent of the non compressed graph , . Out of this graph , transformation 3 generates the graph $,_{\text{red}}$ which is transformed into the graph $,_v$ by transformation 4. *Transformation N* generates from the graph , the graph $,'$:

$$, \xrightarrow{\text{Tr.3}} ,_{\text{red}} \xrightarrow{\text{Tr.4}} ,_v \qquad\qquad , \xrightarrow{Tr.N} ,'$$

We have to show now that $,_v = ,'$. To make the proof easier we assume the following variant of transformation 4 :

$$\begin{array}{rcl} \forall A, B \in Z[,_{\text{red}}] & : & p \in N^*(A,B) \\ & \Longleftrightarrow & A \xrightarrow{r} B \in ,_v \qquad \text{with } r = \prod_{1 \leq i \leq |p|} \mid r(p_i) \mid \end{array}$$

Furthermore we will consider only graphs that fulfil the consistency condition.
Transformation N is then of the following form:

/1/ $U \longleftarrow \{S\}; Z \longleftarrow \{\}; \Gamma' \longleftarrow \{\};$
/2/ while $U \neq Z$ do
/3/ choose any state $A \in U \setminus Z$
/4/ for all $p \in N^*_\Gamma(A) \stackrel{def}{=} \cup_{B \neq A} N^*_\Gamma(A, B)$ do
/5/ $\Gamma' \longleftarrow \Gamma' \cup \{A \stackrel{\mathrm{r}}{\longrightarrow} z(p_{|p|})\}$ with $r = \mid \prod_i r(p_i) \mid$
/6/ $U \longleftarrow U \cup \{z(p_{|p|})\}$
/7/ done
/8/ $Z \longleftarrow Z \cup \{A\}$
/9/ done

This algorithm terminates because both the inner and the outer loop terminate after a finite number of steps.
The proof that $\Gamma' = \Gamma_v$ is done in two steps:

1. $\Gamma' \subseteq \Gamma_v$
2. $\Gamma_v \subseteq \Gamma'$

1. Proof that $\Gamma' \subseteq \Gamma_v$

Assertion 1: when the algorithm terminates we have:

$$\Gamma' \subseteq \Gamma_v \tag{1}$$
$$U \subseteq Z[\Gamma_v] \tag{2}$$
$$U \subseteq Z[\Gamma_{\mathrm{red}}] \tag{3}$$

Proof of assertion 1 : before line /2/ is executed (1), (2) and (3) are fulfiled.

Assertion 2 : when (1), (2) and (3) are fulfiled before line /3/ is executed then (1), (2) and (3) are also fulfiled after the execution of line /8/

Proof of assertion 2 : (1) - (3) are fulfiled before line /3/ is executed. In line /5/ p is a path in Γ between A and a successor* state $B = z(p_{|p|})$. Because of $A \in U$ we have because of (3): $A \in Z[\Gamma_{\mathrm{red}}]$ and therefore $p \in \Psi[\Gamma_{\mathrm{red}}]$. B is stable in Γ and therefore also in Γ_{red} and so we have:

$$A \stackrel{r}{\longrightarrow} B \in \Gamma_v \qquad \text{with} \qquad r = \prod_i \mid r(p_i) \mid$$

After the execution of line /5/ (1) is fulfiled. Because of $B \in \Gamma_v$ and $B \in \Gamma_{\mathrm{red}}$ (2) and (3) are fulfiled after the execution of line /6/.
Therefore after the algorithm terminates (1), (2) and (3) are fulfiled.

2. Proof that $\Gamma_v \subseteq \Gamma'$

Note: For all stable states B in Γ with $S \stackrel{*}{\longrightarrow} B$ the state A in line /3/ takes on the value B in some loop run.

Let now $C \xrightarrow{r} D$ be an arc of $,_v$ then we have to prove that $C \xrightarrow{r} D \in ,'$.

$$C \xrightarrow{r} D \in ,_v \quad \Longrightarrow \quad \exists p \in \Psi[,_{\text{red}}] : p \in N^*_{\Gamma_{\text{red}}}(C, D) \qquad \text{with } r = \prod_i \mid r(p_i) \mid$$

Because of transformation 3 only removes arcs we have : $p \in \Psi[,]$. C and D are stable in , because , is homogenuous and therefore we have $p \in N^*_\Gamma(C, D)$. Furthermore we have for $,_{\text{red}} : S \xrightarrow{*} D$. Because of the note above we have that A takes on the value C in line /3/ and therefore we have after the execution of line /7/ (because D is a successor* state of C in ,):

$$C \xrightarrow{r} D \in ,'$$

and therefore:

$$C \xrightarrow{r} D \in ,_v \qquad \Longrightarrow \qquad C \xrightarrow{r} D \in ,'$$

Therefore we have $,_v = ,'$ and *transformation N* is functionally equivalent to the transformation steps 3 and 4.

2.4 Computation of the state probabilities

Now the state probabilities of the Markovian model are computed by solving the system of equations $p\ Q\ = 0$ where Q is the generator matrix.
The following methods are provided:

- *(1) power*
- *(2) power2*
- *(3) lpu*
- *(4) crout*
- *(5) grassmann*

The five names represent the following methods ([HENN88]):

1. **power:** Jacobi method for matrices with dimension <= 65000. The Jacobi method is an iterative method. A upper limit for the number of iterations must be specified. This number must be typed in interactively. To reach high precision , more iterations are necessary than in *power2* [STEW78].

2. **power2:** Gauss Seidel method for matrices with dimension <= 65000. The Gauss Seidel method is an iterative method. The idea of this method is to start with a startvector and to compute a series of vectors that converge to the solution vector. An upper limit for the number of iterations must be specified. This number must be typed in interactively. [STEW78]. The advantage of this iterative methods is that the iteration can be stopped if the difference between two successive solution vectors is smaller than ϵ. In this way you can choose between the accuracy of the solution and the computation time.

3. **lpu:** The LPU method is a direct method.

4. **crout:** Crout method for matrices with dimension <= 1000. The Crout method is only suitable for small Markovian models with dimension less than 1000. The demand for memory grows quadratically in the dimension of the problem to solve. It is a direct method.

5. **grassmann:** Grassmann method. This method is a variant of the Gauss-elimination method. The idea of this method is to compute the diagonal elements of the matrix $\underline{Q}$ as the sum of the non diagonal elements of a column. Overflow problems can occur, if the dimension of the matrix is too big. The power of the algorithm is not equal to *power*, *power2* and *lpu* when analyzing big systems. ([MUNK91])

3 The Model Description Language MOSLANG

The language MOSLANG consists of a series of constructs which can be summarized in four parts, the declaration part, the vector description part, the rules part and the results part.
In this chapter the semantic of each part is briefly explained.

Declaration part: This part contains the declarations. Symbolic names can be given to each component of the state vector. Constants can be defined as in C. All parameters and the required performance measures must be declared here. The type and name of the parameter or performance measure must be specified here.

Vector description part: In this part the dimension and the range of the state space is fixed. Also the prohibited states are specified.

Rules part: In the rules part the state transitions are specified. This can be done by specifying a number of RULE constructs.
The RULE construct consists of a condition and an action part. If all conditions are fulfiled, the specified actions will be executed.
The action part also contains the transition rate and the transition probability.
One RULE construct specifies several state transitions. It is not necessary to specify an individual RULE construct for every state transition.
It is possible to use *immediate transitions*. These are executed immediately, when the conditions are fulfiled. Thus modeling preemption and priorities is simple. Immediate transitions are characterized by the transition rate –1.0.
By using immediate transitions one achieves a decoupling of the different nodes of the system. This causes a very powerful compression of the model description.

Results part: In the results part the performance measures are specified.
The results part consists of RESULT constructs, RESULT MEAN constructions, RESULT DIST constructs and FINAL constructs. In the RESULT construct the basic performance measures which can be deduced directly from the state probabilities are computed. In the RESULT MEAN construct and in the RESULT DIST construct the average number of jobs in a station and the probability of each possible queue length in a station can be determined. In the FINAL construct performance measures which are functions of the basic performance measures and the system parameters, for example average waiting times, utilization, throughputs, can be computed.

4 Application of the Markov Analyzer MOSES

In this chapter the instructions for using the program package MOSES are given. Furthermore the semantic of the single constructions of the specification is explained.

4.1 Simple Example

To make the usage of *MOSES* clear a simple example of a fault tolerant multiprocessor system will be specified using MOSLANG. Within the example the user has to fill in the sequences in acute brackets $< >$. The rest of the specification will be added automatically. $\ll$ *return* $\gg$ means that the user has to press return.

4.1.1 Characterization of the System

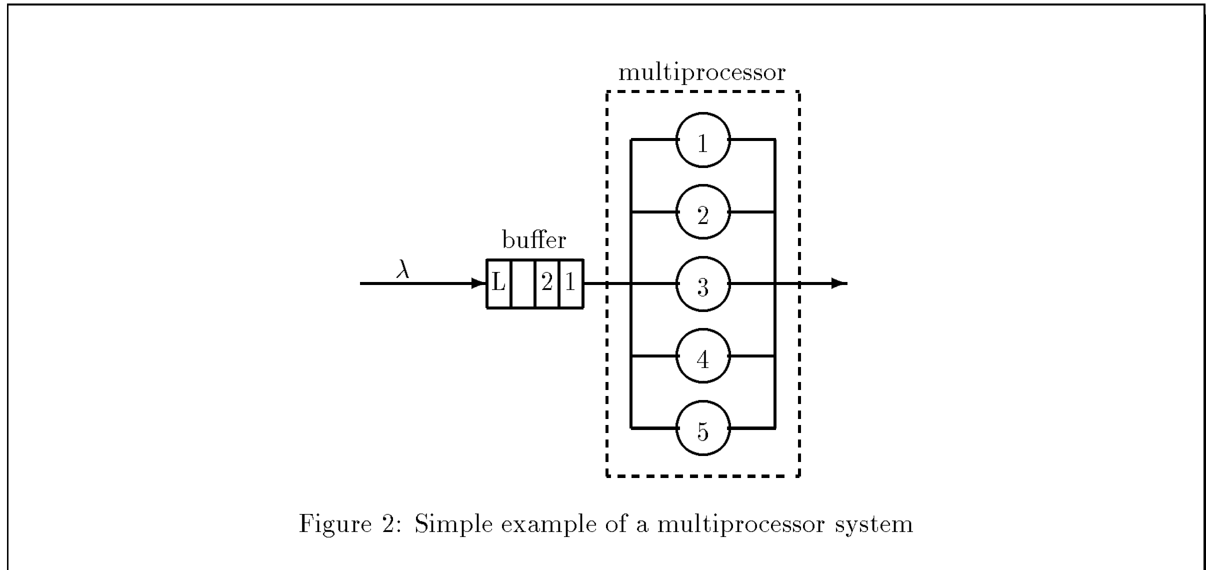

Figure 2: Simple example of a multiprocessor system

In Fig. 2 it is shown a simple example of a multiprocessor system with finite arrival buffer. Failure of servers is possible. It is assumed that after a server failure the server is repaired again.

- the system has a finite queue length L.
- the arrival rate at the queue is λ.
- the time between the failure of two server is $mtbf$.
- the time to repair a failed server is $mttr$.
- if a server that is serving a job fails and there is still a server free then this job is executed again by the new server.
- if a server that is serving a job fails and no other server is free then the job has to wait until the server is repaired or another server becomes free.

4.1.2 Define block

If the user has choosen *(1) define block* in the above described menue and if he has affirmed to use *defines*, then

- *# define*
- VAL[0]
- VAL[1]
- VAL[2]

appear on the screen and the user can choose symbolic identifiers for each component of this three dimensional state space.

4.1.3 Definition of constants

With the help of the *#define* construct it is also possible to define constants as in normal C syntax. In our multiprocessor example we do not use constants.

4.1.4 VAR block

A special VAR construct has to be specified for each variable that shall be declared. At first the user is asked for the type of the variable. Possible types are

- i (int): the variable is an integer.
- d (double): the variable is real.
- p (prob): the variable is a real value between 0 and 1. It represents a probability.

Subsequently the user is asked for the name of the corresponding variable. All correct C-identifiers can be used as variable names.

4.1.5 OUTPUT block

For each performance measure that shall be computed one has to declare an OUTPUT construct. The input for an OUTPUT construct is analogous to the input of a VAR construct.

4.1.6 DIM block

Here the dimension of the state space vector is determined.

4.1.7 NOT construct

With the help of this construct the prohibited states are specified. That means that this states are not allowed to appear in the Markov chain. If any state in the state space fulfils the NOT construct then it is a prohibited state for the specified system.
The user can specify as many NOT constructs as necessary. A NOT construct consists of one or more predicates and the whole predicate is true when *all* seperate predicates of the NOT construct are true.

4.1.8 RULE construct

In this part of the specification the rules which determine the behaviour of the system are specified. A RULE construct consists of a condition part and an action part.
The condition part consists of one or more predicates.
The action part describes the state transition in the Markov chain. It consists of one or more actions, the transition rate and a transition probability. The actions in the action part are executed when *all* specified predicates in the condition part are fulfiled.
Using -1.0 as transition probability indicates a timeless (immediate) transition.

4.1.9 RESULT construct

In this part of the specification the basic performance measures which can be derived directly from the state probabilities are computed. The RESULT construct consists of a condition part and an action part.
The condition part again consists of predicates. If all predicates are fulfiled the performance measures named in the action part will be computed by executing the instructions specified there.
The performance measures must be declared in the OUTPUT part of the specification.
All performance measures that have to be computed must be declared in the OUTPUT block and mentioned again in the so called FINAL construct.

4.1.10 RESULT MEAN construct

With the help of this construct the average queue length in the nodes of the system can be computed. This construct as well as the RESULT DIST construct is not used in the simple multiprocessor specification.

4.1.11 RESULT DIST construct

With this construct the state probabilities of each possible state in the Markov chain can be computed.

4.1.12 FINAL construct

Performance measures which shall appear on the screen or in the *Results* file must be specified using one FINAL construct for each performance measure.

After all FINAL constructs have been specified and the question *Yet another FINAL? y/n* has been answered with 'n', the specification is complete.

4.2 Specification of the fault tolerant multiprocessor system

The following specification of the system is stored in the specification file.

```
#define   queue     VAL[0]      /* number of jobs in the queue */
#define   active    VAL[1]      /* number of active jobs */
#define   working   VAL[2]      /* number of non defect CPU's */

VAR       int       L;          /* finite capacity of the queue */
VAR       double    mue ;       /* service rate of each CPU */
VAR       double    lambda ;    /* arrival rate of jobs at the queue */
VAR       double    coverage ;  /* coverage factor */

VAR       double    mtbf ;      /* mean time between failure */

VAR       double    mttr ;      /* mean time to repair */

OUTPUT    double    qquer ;     /* average queue length */

DIM     3;
MIN     0       0       0 ;
MAX     L       5       5 ;

NOT { queue + active > L + 5 } ;

/* Rule for the arrival of jobs at the system */

RULE { queue < L } -> queue += 1 : lambda : 1.0 ;

/* Rule for the failure of a component */

RULE {working > 0 } -> working -= 1 : 1/mtbf : 1.0 ;

/* Rule for repairing a failed component */

RULE {working < 5 } -> working += 1 : 1/mttr : 1.0 ;

/* At least one server is free to service a job */
```

```
RULE { queue > 0 }{ working - active > 0 } ->
          active += 1 : -1.0 : 1.0 ;

/* Rules for finishing jobs */

RULE { active == 1 }    { working >= 1 } ->
          active -= 1 : 1*mue : 1.0 ;
RULE { active == 2 }    { working >= 2 } ->
          active -= 1 : 2*mue : 1.0 ;
RULE { active == 3 }    { working >= 3 } ->
          active -= 1 : 3*mue : 1.0 ;
RULE { active == 4 }    { working >= 4 } ->
          active -= 1 : 4*mue : 1.0 ;
RULE { active == 5 }    { working == 5 } ->
          active -= 1 : 5*mue : 1.0 ;

RESULT  qquer    =    MEAN queue ;
```

4.2.1 Graphical representation of the results

For the plots the following model parameters are assumed

L	=	10
mue	=	0.1
lambda	=	0.5
mtbf	=	0 .. 1000
mttr	=	50

The coverage factor is varied to show the effect of this factor.

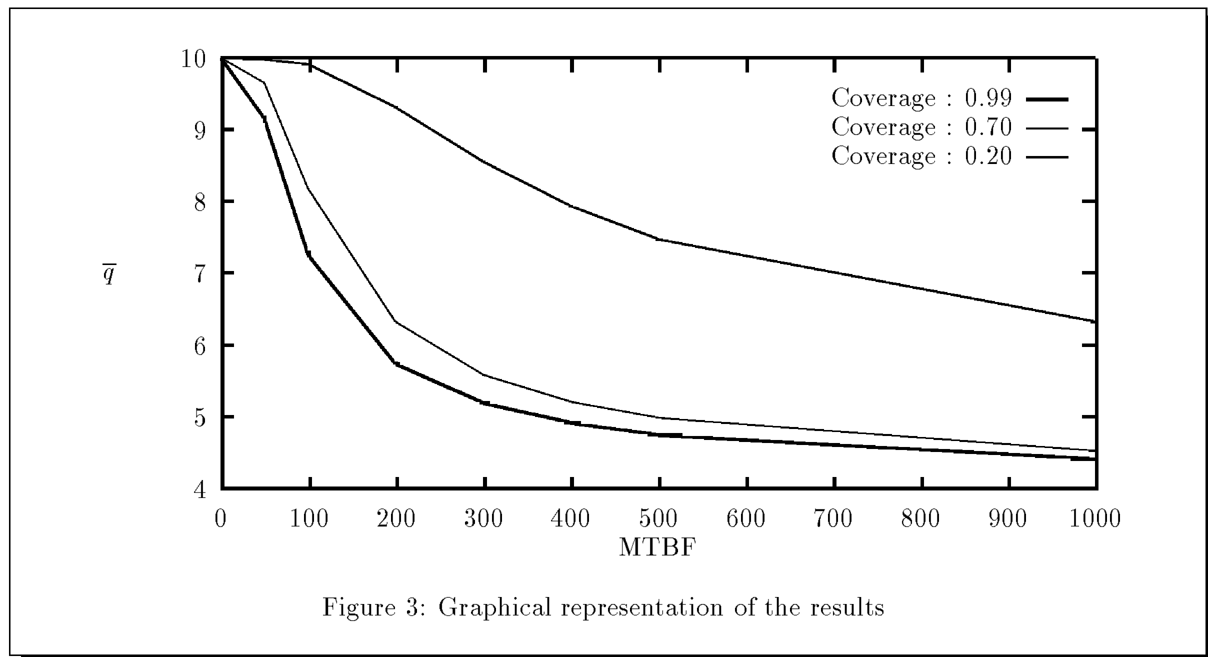

Figure 3: Graphical representation of the results

As can be seen the higher the mean time between failure the lower is the average queue length. The limit for mtbf to infinite is 4.07.

5 Specification of a Fault Tolerant UNIX System Using MOSLANG

In the former chapter the usage of MOSES was shown on a very simple fault tolerant multiprocessor system. In this chapter it is shown how MOSES can be used to specify the much more complicated model of a fault tolerant UNIX system. The model of this system is described in [JUNG91].

5.1 Description of the model

5.1.1 Model of a job

The typicle live cycle of a job in a UNIX system can be described with four states [JUNG91:

(a) a user job is executed (*user*).

(b) a system call is handled (*kernel*).

(c) a driver routine is executed (*driver*).

(d) waiting for the end of an I/O demand (*io*).

5.1.2 Characterization of the system

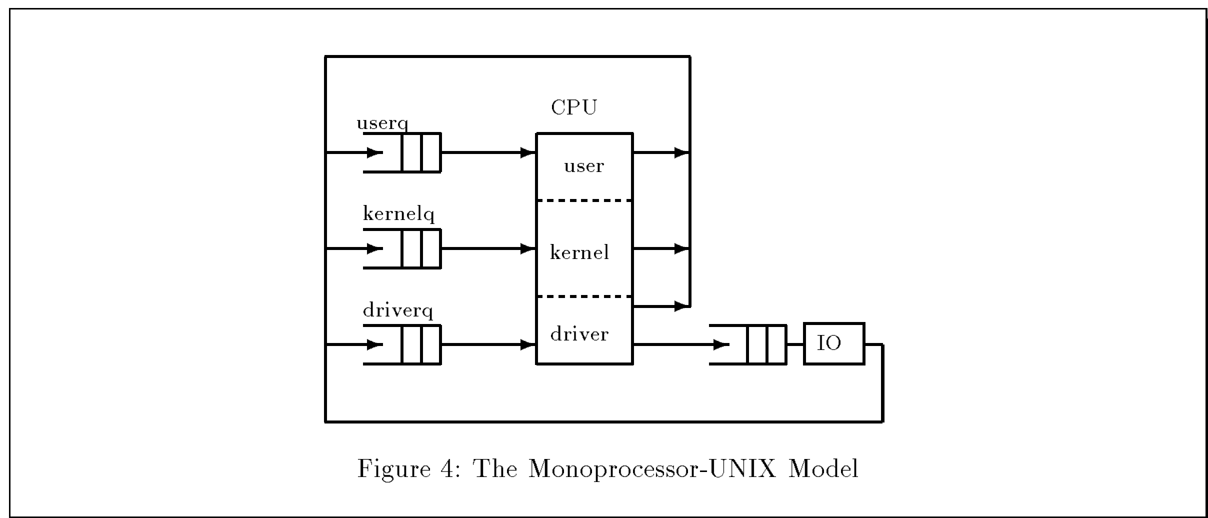

Figure 4: The Monoprocessor-UNIX Model

The model of the monoprocessor system is shown in Fig. 4 The system consists of three queues *userq kernelq driverq* and the four stations *user kernel driver* and *io*. The live cycle of a job (see Fig. 5) always starts in the kernel context and if the job is finished it will be replaced by a new job. That means that the number of jobs in the system is always constant (closed queueing network).
We make the following assumptions :

(a) jobs can not be distinguished from each other.

(b) jobs working in the context *user* can be preempted by jobs in the context *kernel* and *driver*. In all other contexts no preemption is possible.

(c) we assume the following priorities:
driver > kernel > user.

(d) the service times in the contexts *kernel* and *driver* are distributed exponentially with the average service times *kerneltime* and *drivertime*.

(e) the periphery is assumed to be load dependent with exponentially distributed service times *IOTIME(n)*, depending on the number of jobs in that station.

(f) the system is in the stationary state.

(g) the time between two CPU failures is exponentially distributed with rate *1/MTBF_CPU*.

(h) the time to repair the CPU is exponentionally distributed with rate *1/MTTR_CPU*.

(i) the time between two IO failures is exponentially distributed with rate *1/MTBF_IO*.

(j) the time to repair the IO is exponentially distributed with rate *1/MTTR_IO*.

(k) the coverage factor of the CPU is *coverage_CPU*.

(l) the coverage factor of the IO is *coverage_IO*.

(m) if the CPU crashes then the job that is just in service on the CPU is started again when the CPU is repaired.

(n) if the IO crashes and a job is just in service on the IO then this job is started again when the IO is repaired.

5.2 Specification of the model

The specification of the model can be seperated into three parts

1. determining the possible locations and the possible states of the jobs.
2. determining the valid states of the model.
3. determining the possible state transitions and transition rates.

5.2.1 Possible locations of a job

In our system a job can be in one of the following locations

(a) a job is processed in the CPU

(b) a job waits for completion of an I/O demand (*io*).

(c) a job waits for handling an I/O demand (*driverq*).

(d) a job waits for CPU time in kernel context (*kernelq*).

(e) a job waits for CPU time in user context (*userq*).

(f) jobs are waiting and the CPU is down.

(g) jobs are waiting and the IO is down.

The CPU can be *idle*, working in user context (*user*), working in kernel context (*kernel*) or working in driver context (*driver*) . The state of *io*, *driverq* and *userq* can be described by the number of jobs waiting there. Furthermore one has to distinguish between the states *UP_CPU, DOWN_CPU* and *UP_IO, DOWN_IO*. Therefore we have a 7-dimensional state space of the form :

$$(\mathrm{cpu}, \mathrm{userq}, \mathrm{kernelq}, \mathrm{driverq}, \mathrm{io}, \mathrm{state_CPU}, \mathrm{state_IO})$$

The first part of the specification file is the definition of the seven states of the state space. This is done by the *#define* construct in the following way :

```
/* Monoprocessor model */
#define   cpu        VAL[0]   /* state of CPU */
#define   userq      VAL[1]   /* # jobs waiting userkontext service */
#define   kernelq    VAL[2]   /* # jobs waiting kernelkontext service */
#define   driverq    VAL[3]   /* # jobs waiting driverkontext service */
#define   io         VAL[4]   /* # jobs waiting IO completion */
#define   state_CPU  VAL[5]   /* state of the CPU */
#define   state_IO   VAL[6]   /* state of the IO */
```

In the *#define* part one can also define constants as in normal C syntax.

```
/* state space of CPU */
#define   idle        0    /* CPU idle */
#define   user        1    /* CPU in state user */
#define   kernel      2    /* CPU in state kernel */
#define   driver      3    /* CPU in state driver */
#define   up_CPU      1    /* CPU is operational */
#define   down_CPU    0    /* CPU is defect */
#define   up_IO       1    /* IO is operational */
#define   down_IO     0    /* IO is defect */
```

5.2.2 Determination of the valid states

To determine the valid states of the system it is usefull to exclude the prohibited states from all possible states of the system.

The following conditions must not be violated :

1. if the CPU is in the state *idle*, the number of jobs in all other possible locations of the system is K.
2. if the CPU is not *idle*, the number of jobs in all other possible locations of the system is K–1.

This conditions can be formulated in MOSES as follows :

```
NOT {cpu == idle} {userq + kernelq + driverq + io != K} ;
NOT {cpu != idle} {userq + kernelq + driverq + io != K-1} ;
```

5.2.3 Determination of the possible state transitions

The normal process behaviour can be described as follows :

(a) context switch from *user* to *kernel* (begin of a system call).
(b) context switch from *kernel* to *user* (end of a system call).
(c) context switch from *kernel* to *driver* (an I/O demand has occured).
(d) context switch from *driver* to *kernel* (an I/O demand has been finished).
(e) changing from *driver* to *io* (start serving an I/O demand).
(f) changing from *io* to *driver* (end of serving an I/O demand).
(g) a job finishes and is replaced by a new one.

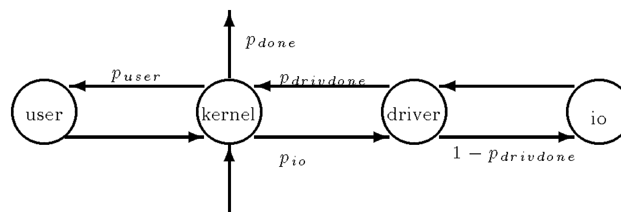

Figure 5: The model of a job in the UNIX system

The state transition probabilities are shown in Fig. 5 where the model of a job in the UNIX-system is shown.

5.2.4 Rules for the failure and repair of components

At first we will show how th specify the failure and repair of components.

```
RULE {state_CPU == up_CPU} ->
        state_CPU == down_CPU : 1/MTBF_CPU : 1.0 ;
RULE {state_CPU == down_CPU} ->
        state_CPU == up_CPU : 1/MTTR_CPU : coverage_CPU ;
```

The specification of the failure and repair rules of the IO is analogous.

5.2.5 Rules for the preemption mechanism

Jobs processed in the user context can be preempted from the CPU if the CPU is operational and if there is a job waiting in the kernel context or driver context. This is the case when one of these queues is not empty. Furthermore we have to keep in mind that jobs in the context driver have higher priority than jobs in the context kernel.
The behaviour of the job can therefore be specified in the following way :

```
RULE {cpu == user} {driverq > 0 } {state_CPU == up_CPU } ->
        cpu = driver driverq -= 1 userq += 1 : -1.0 : 1.0 ;
RULE {cpu == user} {driverq == 0 } {kernelq > 0 } {state_CPU == up_CPU }->
        cpu = kernel kernelq -= 1 userq += 1 : -1.0 : 1.0 ;
```

If the operational CPU is in the context *user* and the driver queue is not empty then the job working in the context user is preempted and added to the user queue. Then the context of the CPU is changed into the context *driver* and the number of jobs in the driver queue is reduced by one.
It is assumed that this context switch is timeless (i.e. the corresponding transition rate is -1.0).
In order to indicate that driver routines have the highest priority it is only necessary to mention that *driverq* > 0. That means that if jobs are waiting in the driver queue they will get the CPU.

The second rule specifies that if the operational CPU is in the context *user* and if no job is waiting in the driver queue then a job waiting in the kernel queue can preempt the user job in the CPU. In order to make sure that a kernel job can not preempt a driver job one has to write the condition *driverq* $== 0$. It is again assumed that the state transition is timeless.

5.2.6 Rules for the priority mechanism

Till now we have only specified the preemption mechanism. What we have to specify now is the priority mechanism.
The fact that driver routines have the highest priority is expressed by the following rule :

```
RULE {cpu == idle} {driverq > 0 } {state_CPU == up_CPU } ->
        cpu = driver driverq -= 1 : -1.0 : 1.0 ;
RULE {cpu == idle} {driverq == 0 } {kernelq > 0 } {state_CPU == up_CPU } ->
        cpu = kernel kernelq -= 1 : -1.0 : 1.0 ;
RULE {cpu == idle} {driverq == 0 } {kernelq == 0 } {userq > 0 } {state_CPU == up_CPU }  ->
        cpu = user userq -= 1 : -1.0 : 1.0 ;
```

If the CPU is idle and a job is waiting in the driver queue then the job gets the CPU no matter how many jobs are in the other queues. If the driver job enters the CPU the number of jobs in the driver queue is reduced by one.
If the driver queue is empty and a job is waiting in the kernel queue then the kernel job can enter the idle CPU. Again the corresponding number of jobs in the kernel queue is reduced by one.
Only if no driver jobs and no kernel jobs are waiting in the queues then a user job can enter the CPU.

5.2.7 Rules for the behaviour of the IO

Finally one has to specify the behaviour of the peripheral system which is done in the specification of the complete system .

5.2.8 Specification of the performance measures

In this example we may be interested for example in the throughput of the system. This can be specified by the construct

```
RESULT {CPU == kernel}    {state_CPU == up_CPU} -> rho_cpu_kernel += PROB ;
RESULT MEAN    userq ;
FINAL   lambda    = 1000 * (pdone * rho_cpu_kernel / kerneltime);
```

but it is also possible to compute all other performance measures of the system.

5.3 Specification of the model

After all this explanations the whole specification of the system is shown below.

```
/* Monoprocessor model */
#define   cpu         VAL[0]     /* state of CPU */
#define   userq       VAL[1]     /* # jobs waiting userkontext service */
#define   kernelq     VAL[2]     /* # jobs waiting kernelkontext service */
#define   driverq     VAL[3]     /* # jobs waiting driverkontext service */
#define   io          VAL[4]     /* # jobs waiting IO completion */
#define   state_CPU   VAL[5]     /* state of the CPU */
#define   state_IO    VAL[6]     /* state of the io */

#define   idle        0      /* CPU idle */
#define   user        1      /* CPU in state user */
#define   kernel      3      /* CPU in state kernel */
#define   driver      4      /* CPU in state driver */
#define   up_CPU      1      /* CPU is operational */
#define   down_CPU    0      /* CPU is defect */
#define   up_IO       1      /* IO is operational */
#define   down_IO     0      /* IO is defect */

/* modelparameter */
VAR   int      K;             /* # of jobs circulating */
VAR   double   usertime;      /* average time between two system calls */
VAR   double   kerneltime;    /* average time for a system call */
VAR   double   drivertime;    /* average time for a driver routine */
VAR   double   MTBF_CPU;      /* mean time between failure CPU */
VAR   double   MTBF_IO;       /* mean time between failure IO */
VAR   double   MTTR_CPU;      /* mean time to repair the CPU */
```

```
VAR    double  MTTR_IO;     /* mean time to repair the IO */
VAR    double  coverage_CPU;/* coverage factor of the CPU */
VAR    double  coverage_IO; /* coverage factor of the IO */
VAR    prob    pdone;       /* probability that a job is done */
VAR    prob    pio;         /* probability that an IO demand occurs */
VAR    prob    pdrivdone;   /* probability that an IO demand finishes */

OUTPUT   double    rho_cpu_kernel;      /* utilization of the CPU in the context kernel */
OUTPUT   double    lambda;              /* throughput of the system */

/* state space */
/* cpu userq kernelq driverq io stata_CPU state_IO */
DIM   7;
MIN   idle     0     0     0     0     down_CPU     down_IO ;
MAX   driver   K     K     K     K     up_CPU       up_IO;

/* prohibited states */
NOT {CPU == idle} {userq + kernelq + driverq + io != K};
NOT {CPU != idle} {userq + kernelq + driverq + io != K-1};

/* rules for the failure of components */
RULE {state_CPU == up_CPU} ->
       state_CPU = down_CPU : 1.0/MTBF_CPU : 1.0;
RULE {state_IO == up_IO} ->
       state_IO = down_IO : 1.0/MTBF_IO : 1.0;


/* rules for repairing the components */
RULE {state_CPU == down_CPU} ->
       state_CPU = up_CPU : 1.0/MTTR_CPU : coverage_CPU;
RULE {state_IO == down_IO} ->
       state_IO = up_IO : 1.0/MTTR_IO : coverage_IO;


/* I/O behaviour */
RULE {io > 0}   {state_IO == up_IO}   -> io -= 1 driverq += 1 :
  io == 1 ? 1.0 / 28.00,
  io == 2 ? 1.0 / 18.667,
  io == 3 ? 1.0 / 15.5553,
  io == 4 ? 1.0 / 13.998,
  io == 5 ? 1.0 / 13.0668,
  io == 6 ? 1.0 / 12.4443,
  io == 7 ? 1.0 / 11.99991,
  io == 8 ? 1.0 / 11.6669,
  io == 9 ? 1.0 / 11.4037,
  11.20 : 1.0;

/* normal behaviour of a process */
RULE {cpu == user}   {state_CPU == up_CPU} ->
       cpu = kernel : 1.0/usertime : 1.0;
RULE {cpu == kernel}   {state_CPU == up_CPU} ->
       cpu = user : 1.0/kerneltime : 1.0 - pio - pdone;
RULE {cpu == kernel}   {state_CPU == up_CPU} ->
       cpu = driver : 1.0/kerneltime : pio;
```

```
RULE {cpu == kernel}   {state_CPU == up_CPU} −>
        cpu = idle kernelq += 1 : 1.0/kerneltime : pdone;
RULE {cpu == driver}   {state_CPU == up_CPU} −>
        cpu = idle kernelq += 1 : 1.0/drivertime : pdrivdone;

RULE {cpu == driver}   {state_CPU == up_CPU}   {state_IO == up_IO} −>
        cpu = idle io += 1 : 1.0/drivertime : 1 - pdrivdone;

/* preemption */
RULE {cpu == user}   {driverq > 0}   {state_CPU == UP_CPU} −>
        cpu = driver driverq −= 1 userq += 1 : -1.0 : 1.0;
RULE {cpu == user}   {driverq == 0}   {kernelq > 0}   {state_CPU == UP_CPU} −>
        cpu = kernel kernelq −= 1 userq += 1 : -1.0 : 1.0;

/* priorities */
RULE {cpu == idle}   {driverq > 0}   {state_CPU == UP_CPU} −>
        cpu = driver driverq −= 1 : -1.0 : 1.0;
RULE {cpu == idle}   {driverq == 0}   {kernelq > 0}   {state_CPU == UP_CPU} −>
        cpu = kernel kernelq −= 1 : -1.0 : 1.0;
RULE {cpu == idle}   {driverq == 0}   {kernelq == 0}   i {userq > 0}   {state_CPU == UP_CPU} 1−>
        cpu = user userq −= 1 : -1.0 : 1.0;

/* results */


RESULT MEAN userq;
RESULT {CPU == KERNEL}   {state_CPU == UP_CPU}   −>   rho_cpu_kernel += PROB;
FINAL lambda = 1000 * (pdone * rho_cpu_kernel / kerneltime);
```

5.4 Graphical representation of the results

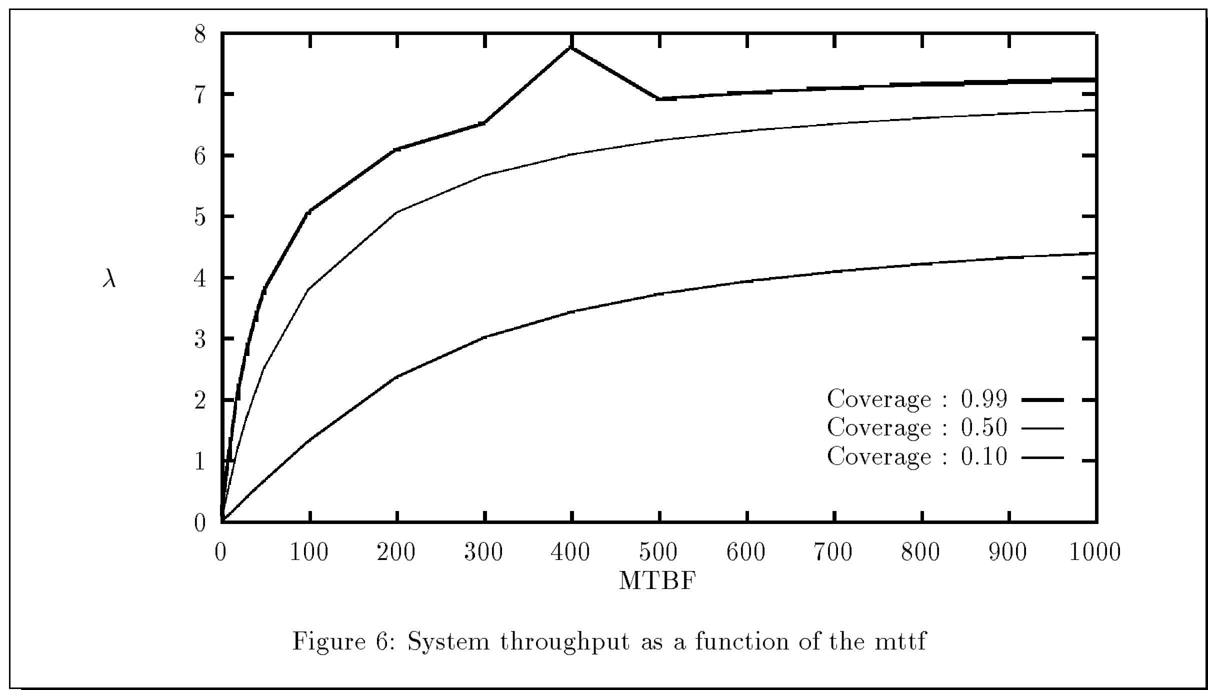

Figure 6: System throughput as a function of the mttf

As can be seen, the throughput of the system increases when the mean time between failure increases. This is exactly the expected behaviour.

6 Conclusion

MOSES (together with the specification language MOSLANG) is a very powerful tool for specifying and analyzing complicated systems. It is very easy to use and if the user is familiar with the tool he can also specify his system without using the program *input.spec* by writing the specification directly with the editor. At the moment MOSES provides only steady state analysis of systems but it is planed to extend the tool to transient analysis.
In this paper it is shown how to describe two fault tolerant models with MOSLANG. The real power of MOSES becomes clear when looking at the example of the fault tolerant UNIX model. The state space of this model with 30 jobs in the system consists of 40734 states and 143713 edges. It is absolutely impossible to construct this Markov chain by hand. But with **MOSES** it is very easy to describe the system in a very compact manner and to create the Markov chain out of this description. This UNIX model can also be extended to a multiprocessor UNIX model with parallel kernel (called *Associated Processors model*) and to a multiprocessor UNIX model with nonparallel kernel (called *Master Slave model*) [JUNG91]. It is relatively simple to describe this complex models (class switching of the jobs, mixed priority strategy) with MOSLANG.
With the description language MOSLANG it is possible to describe e.g. fault tolerant systems, queueing systems, manufactureing systems, Markov-reward models, precedenc graphs, Petri nets and lots of other systems. To make the description of a system easier for the user it is planed to extend the tool with makros that contain the most common features of each of the above described types of systems. Furthermore it is planned to include new techniques for creating the generator matrix and solving the system of equations.

References

[BOLCH89] Bolch, G.: *Leistungsbewertung von Rechensystemen mittels analytischer Warteschlangenmodelle*, Teubner, Stuttgart, 1989.

[CHUNG67] Chung, Kai L.: *Markov Chains*, Springer, Berlin, 1967.

[DYNK65] Dynkin, E. B.: *Markov Processes*, Springer, Berlin, 1965.

[GREI93] Greiner S.: *Leistungsbewertung des Betriebssystems UNIX mit Hilfe approximativer analytischer Verfahren*, Studienarbeit am IMMD IV, Erlangen 1993.

[HENN88] Hennecke, C.: *Parallele Implementierung der numerischen Analyse*, Diplomarbeit am IMMD IV, Erlangen, 1988.

[JUNG91] Jung, H.: *Leistungsbewertung UNIX-basierter Betriebssysteme für Multiprozessoren mit globalem Speicher*, Dissertation FAU Erlangen-Nürnberg, 1991.

[MENN89] Menninger, P.: *Analyse Markov'scher Zustandsräume mit instabilen Zuständen*, Studienarbeit am IMMD IV, Erlangen, 1989.

[MUNK91] Munkert, F.: *Effiziente Erzeugung von Generatormatrizen für zeitkontinuierliche Markovketten*, Studienarbeit am IMMD IV, Erlangen, 1991.

[RACK91] Rackl, K. H.: *Entwurf und Implementierung einer Sprache zur Berechnung von Leistungsgrößen Markov'scher Modelle*, Studienarbeit am IMMD IV, Erlangen, 1991.

[STEW78] Stewart, W. J.: *A Comparison of Numerical techniques in Markov Modelling*, Communications of the ACM, Vol. 21, pp. 144–152, No. 2 feb. 1978.

[WILK72] Wilkinson, J. H.: *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1972.