Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A QUERY LANGUAGE FOR MARKOV CHAINS: COMPILATION, OPTIMIZATION, AND EXECUTION

S. Berson
R. Muntz

September 1991
CSD-910067

A Query Language for Markov Chains: Compilation, Optimization, and Execution*

S. Berson
R. Muntz
Computer Science Department
UCLA

August 29, 1991

Abstract

Queries are one of the most important parts of any modeling system as it is through queries that users express which "results" of the model they desire. Unfortunately, these queries are very dependent on the exact model being studied. This dependence of query on model can make specifying a query a painful process that has to be repeated for each query on every model studied. The usual remedy to this problem is to provide a very specific language for any anticipated queries, rather than a general language. This approach has the drawback, particularly for transient analysis, that it is very difficult to anticipate those queries that will be necessary. This paper describes a flexible high level query language which greatly simplifies the process of expressing queries. With this language, a user is easily able to specify the patterns of states which form the basis of any query, along with a weight or reward associated with that pattern. Further, we show how to optimize these queries and translate them into a procedural programming language such as C for efficient execution.

*This work was supported by a MICRO grant from the University of California and the Hughes Aircraft Company, by an NSF-CNPq Cooperative Research Grant INT 8902183 and by an equipment grant from Digital Equipment Corporation.

1 Introduction

In the past, most Markov chain tools have largely ignored the issue of a language for querying the results. This paper describes a new query language for Markov chains, including queries for transient measures. This query language is being developed as part of a system for generating and analyzing Markov chains. This system compiles these queries into computation procedures which efficiently turn raw numerical data into useful results.

The most common approach in the past for building Markov chain tools is building a specialized package such as SAVE [GOYA86] for solving reliability and availability models. This type of tool has a very convenient interface for specific types of models. The disadvantage is a lack of flexibility. SAVE has a fixed set of types of analysis measures and only certain types of queries are allowed. Other tools such as SHARPE [SAHN87] and ARIES [MAKA82] are similar in that a particular restrictive set of queries and model constructs are provided.

A much more general approach has been taken by Sanders and Meyer [SAND91] based on stochastic activity networks, a very general form of Petri nets. They consider a variety of measures along with a general way of assigning rewards to individual states. One limitation with Sanders and Meyer is that the actual language based on Petri nets becomes unwieldy as the models become complex. The other limitation is that Sanders and Meyer only provide queries on each individual Markov chain state and do not support queries dealing with sequences of states. Sequences and more generally patterns of states are the most natural way of requesting measures related to the occurrence of transitions. Later in this paper several examples are presented to show how useful this feature is.

1.1 Types of Analytical Measures

In this section, a mathematical description of the different types of numerical analysis measures is reviewed. We deal with systems each of which can be modeled by a continuous time Markov chain $\{X(t), t \geq 0\}$. A reward rate function $r(t)$ is defined to be the instantaneous rate at which reward is being accumulated at time t. The total reward $\mathcal{R}(t)$ accumulated during the time interval $[0, t]$ is the integral of $r(t)$.

$$\mathcal{R}(t) = \int_0^t r(u)du$$

A very general measure of this type, $\mathcal{Y}(s, t)$, is called the distribution of total reward

$$\mathcal{Y}(s,t) = P\left[\mathcal{R}(t) \leq s | X(0)\right]$$

As an example of this type of measure, consider a queueing model with several servers working at different rates. Depending on the arrival process and on the scheduling discipline used, a subset of the servers may be providing service at any given time. Let the reward rate at time t, $r(t)$, be the sum of the service rates of all servers providing service in state $X(t)$. The distribution of total reward $\mathcal{Y}(s,t)$ is the probability that at most s units of service are provided during the time interval $[0,t]$ starting with some initial state probability distribution for $X(0)$.

Another type of measure restricts the set of reward rates to be either zero or one. In this case a function $\hat{r}(t)$ is defined analogous to the function $r(t)$ except that the range of $\hat{r}(t)$ is restricted to 0 or 1. Given a subset of states Ω of the Markov chain which are assigned a reward of 1, three subtypes of queries can be identified: (a) the time spent while $X(t) \in \Omega$ in $[0,t]$, (b) the minimum time t such that $X(t) \in \Omega$, or (c) the number of visits to any state in Ω in $[0,t]$. The random variable $\hat{\mathcal{R}}(t)$, representing the total time spent in states with reward 1, is

$$\hat{\mathcal{R}}(t) = \int_0^t \hat{r}(u)du$$

The distribution of cumulative time $\mathcal{O}(s,t)$ is

$$\mathcal{O}(s,t) = P\left[\hat{\mathcal{R}}(t) \leq s | X(0)\right]$$

As an example application, consider a reliability model where each state is either operational (reward 1) or non-operational (reward 0). The distribution of cumulative (operational) time is the (numerical) distribution of total reward.

The random variable T, representing the time until a selected state (reward 1) is entered for the first time, is called the first passage time.

$$T = \inf_t \{\hat{r}(t) = 1 | X(0)\}$$

The distribution of first passage time $\mathcal{F}(t)$ is

$$\mathcal{F}(t) = P\left[T \leq t | X(0)\right]$$

In a reliability model the measure $\mathcal{F}(t)$ (typically known as system reliability) could be used to find the probability of a system not having a failure within a time interval of length t given $X(0)$.

The last measure in this category, distribution of number of events is slightly different from the previous cases in that it uses impulse rewards rather than reward rates. Impulse rewards differ from reward rates in that they are accumulated at a single time instant as opposed to a reward rate which is accumulated over time. An impulse reward of 1 is earned for each occurrence of the specified patterns[1], e.g.

pattern = s : a reward of 1 is earned each time state s is visited

pattern = x,y where P(x),Q(y) : a reward of 1 is earned for each transition from a state satisfying a predicate P to a state satisfying predicate Q

If $\tilde{\mathcal{R}}(t)$ is the random variable equal to the number of such patterns observed in $(0,t)$, then $\mathcal{N}(k,t)$, the distribution of number of events given $X(0)$

$$\mathcal{N}(k,t) = P\left[\tilde{\mathcal{R}}(t) \leq k | X(0)\right]$$

In a reliability model, the probability of two or less failures in a time interval of length t is an example of this type of measure.

The third group of measures, involving mean reward and mean number of events, are $E[\mathcal{R}(t)]$ and $E[\tilde{\mathcal{R}}(t)]$, respectively.

The last group of measures consists of measures at a specific point in time. Let $P_j(t)) = P[X(t) = j | X(0)]$ and let Z_j be the reward associated with Markov chain state j, then $\mathcal{F}(t)$, the transient point reward is

$$\mathcal{F}(t) = \sum_j Z_j P_j(t)$$

while $\mathcal{S}$, the steady state reward is

$$\mathcal{S} = \lim_{t \to \infty} \sum_j Z_j P_j(t)$$

[1] A full, formal description of pattern specifications is given in the next section

MEASURE	PRF	TIME	THRES	BOUND	INIT
d_total_reward	X	X	X	X	X
d_cumulative_time	X	X	X	X	X
d_first_passage	X	X		X	X
d_number_events	X	X	X	X	X
e_total_reward	X	X		X	X
e_cumulative_time	X	X		X	X
e_first passage	X			X	X
e_number_events	X	X		X	X
point_reward	X	X		X	X
steady_state	X			X	

Figure 1: Supported measures

For each of the types of measures described above, a complete specification requires additional parameters. A *pattern reward function* or a *sequence reward function* is required which specifies when and in what amounts rewards are earned. The pattern or sequence reward function defines the functions $r(t)$, $\hat{r}(t)$, or Z_j used in the above analysis measures and is discussed in the next section. Depending on the analysis measure, additional arguments may be required including an initial state probability distribution, a time interval, an error bound, and a threshold value. The different measures we support and their arguments are shown in Figure 1. The column **MEASURE** is the query name of the different types of numerical measures discussed earlier in this section. These query names will be used in queries to choose which numerical results are desrired. By convention, those measures whose names begin "*d_*" are distributions while those whose names begin "*e_*" are expectations. The other columns represent which arguments are required for each of these measures. The column labeled **PRF** refers to whether a *Pattern Reward Function* (described in the next section) is required. The columns labeled **TIME**, **THRES**, **BOUND**, and **INIT**, refer to whether of not a time horizon, a threshold value, an error bound, and an initial state distribution, respectively are required. As an example, the expected first passage time (e_first_passage) requires a pattern reward function, an error bound, and an initial state distribution.

The remainder of the paper describes our Markov chain query system. The next section describes the high level query language in which queries are expressed. Section three describes how this query language can be translated into a form that can then be executed in a procedural programming language. Section four describes how several of the numerical algorithms evaluate queries and concentrates on several optimization techniques which are important in providing efficient evaluation of queries. The last section presents our

conclusions.

2 Query Language

The rewards and target patterns of a query are specified in a *pattern reward function* each of which has a name and is composed of one or more clauses[2]. The name of the pattern reward function is used to group the clauses of a single function together. A complete query also must refer to a pattern reward function by its name. Each clause of a pattern reward function assigns rewards to patterns of states. For now, pattern reward functions with just one clause are considered. The general structure for each clause of a pattern reward function is

pattern_name(*Pattern_expression*) yields *Reward* where *Predicates*

Intuitively, the meaning of this clause is that the reward *Reward* is assigned to any subsequence of states matching the pattern represented in *Pattern_expression* and satisfying the predicates in *Predicates*. Another more general form for the pattern reward function is known as the *sequence reward function* and has the form

pattern_name(*Sequence_expression*) where *Predicates*

The pattern expression portion of a pattern reward function clause is a regular expression of *pattern states* using a comma "," for the concatenation operator, plus ("+") for the alternation operator, and star ("*") for the transitive closure operator. The sequence expression of a sequence reward function is also a regular expression using the same operators, but allows a reward R to be attached to a pattern state S with a colon, i.e. $S : R$. Regular expressions were chosen for the pattern or sequence reward function for several reasons. The first is that the regular expressions can ultimately be converted to a deterministic finite state automata for efficient calculation. The determinism keeps the computational complexity from getting even more extensive. There are other, more powerful grammars that are also deterministic such as deterministic context free grammars. The advantage that regular expressions provide that other deterministic grammars do not, is the compact and natural representation that regular expressions provide for dealing with sequences of states. Other large classes of deterministic languages require a specification either as a finite state machine (e.g. a deterministic pushdown automata) or via a set of production rules.

[2]The BNF for the language is given in appendix A

The pattern states in the regular expression will match any Markov chain states that satisfy the specified predicates. In the case of sequence reward functions, these pattern states also include a reward that is earned as the sequence of Markov chain states matches the pattern. For example, let $S_1, \ldots, S_n$ be a set of "state variables" called *pattern states* to be used in a pattern expression and let $P_1, \ldots, P_n$ be predicates on these pattern states. $P_1(S_1)$ effectively defines a set of Markov chain states; namely those states that satisfy predicate P_1. Thus, if Markov chain states s_{1a} and s_{1b} are the only two states which satisfy predicate $P_1(S_1)$, then the regular expression S_1^* is equivalent to $(s_{1a} + s_{1b})^*$ (and *not* $s_{1a}^* + s_{1b}^*$). The simplest example of a pattern expression would be S_1, the pattern expression with only one state. The pattern expression S_1 is useful for steady state queries or transient point probability queries. Other queries, especially queries about reliability models, are often based on transitions between Markov chain states. The pattern expression, (S_1, S_2), consisting of two pattern states would be needed for queries about failures. Two predicates would be provided to make pattern state S_1 match operational states and to make pattern state S_2 match non-operational states. Rewards in a sequence reward function are specified by a colon, ":", and a numerical reward value after the pattern state. The simplest non-trivial example of a sequence expression would be $(S_1 : 0 + S_2 : 1)^*$, the sequence expression consisting of any sequence of pattern states S_1 and S_2 where the pattern states S_2 are assigned a reward of 1. The reward for any pattern state whose reward is unspecified is zero. Thus the previous sequence expression can be rewritten $(S_1 + S_2 : 1)^*$. Other examples of pattern and sequence expressions are given later in this section.

Pattern and sequence reward function are not specific to one analysis measure and in general can be reused in multiple queries. Therefore it is desirable that the modeling system allow the user to give the pattern reward function a name and reference it in a query by name. For example, given a pattern or sequence reward function, one might ask for the time until the first occurrence of a nonzero reward, a count of the number of occurrences of the pattern, or the expected total reward earned with this pattern in an interval of time. More specifically, a reward structure for counting the number of failures (in a reliability model) can be used in different queries to calculate the expected number of failures, or the distribution of the number of failures, or even the distribution of first passage time to failure.

The predicates of the pattern or sequence reward function clause form a system of constraints that the pattern states of the pattern or sequence expression must satisfy to earn rewards. While the pattern or sequence expression expresses temporal relationships between different pattern states, the predicates map the pattern states to sets of Markov chain states. These predicates are not limited to just one argument, but rather may have two or more arguments. This will be shown to be useful for queries most easily expressible in terms of transitions (a sequence of two states). As an example consider a reliability model where the user is interested in the number of processor failures. A processor failure is defined as two adjacent states where the latter state has one additional failed processor than the former. The two adjacent states can be expressed in a pattern expression as (S_1, S_2). The predicate

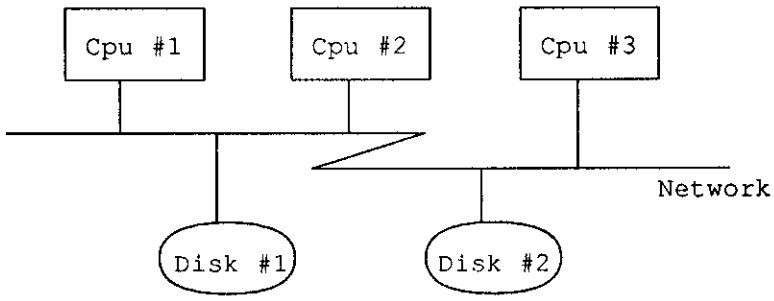

Figure 2: Computer Reliability Model

$P(S_1, S_2)$ on both S_1 and S_2 can be used to easily express the correct relationship.

```
1.  P(S_1,S_2) :-
2.        failed_processors(S_1,X_1),
3.        failed_processors(S_2,X_2),
4.        X2 is X1 + 1.
```

Lines 2 and 3 bind X_1 and X_2 to the number of failed processors in states S_1 and S_2 respectively, while line 4 verifies the appropriate condition. More details on reward functions are given in the next section.

So far, only pattern or sequence reward functions with one clause have been considered. There is an additional issue when pattern or sequence reward functions with multiple clauses are used; namely, *consistency*. Two clauses are defined to be inconsistent if there exists some sequence of states such that the two clauses generate distinct nonzero rewards for the same subsequence. Thus a clause that assigns a reward of 1 to a state S_1 and another clause that assigns a reward of 2 to the same state S_1 are inconsistent. It will be shown that any inconsistency between clauses will be discovered in the process of transforming the query for efficient processing. (This is discussed in section 3). Thus part of the translation process is consistency checking.

2.1 Examples

Now some complete pattern and sequence reward functions will be examined. The first example concerns a reliability model of a repairable computer system. Assume that the system has three processors, two disks, and a bus connecting them. Each component will fail at some (exponential) rate, and each failed component will be repaired at some (exponential) rate. This model is shown graphically in Figure 2. The system is operational if one disk,

one processor and the bus are all operational. The state of this system is represented by the number of operational components and the total number of components of each type. Thus the state for the system in complete working order can be described by the set of 2-tuples $[[cpu, 3/3], [bus, 1/1], [disk, 2/2]]$ where the $[cpu, 3/3]$ means that all three of the cpus are operational. If one *cpu* were to fail, the cpu state would be $[cpu, 2/3]$. The operational and non-operational states can be defined by the following predicates:

operational([[cpu,C/C_0],[bus,B/B_0],[disk,D/D_0]]) :-
C > 0,
B > 0,
D > 0.

non-operational([[cpu,C/C_0],[bus,B/B_0],[disk,D/D_0]]) :- C == 0.
non-operational([[cpu,C/C_0],[bus,B/B_0],[disk,D/D_0]]) :- B == 0.
non-operational([[cpu,C/C_0],[bus,B/B_0],[disk,D/D_0]]) :- D == 0.

We have chosen to use logic programming in the form of Prolog [CLOC87] as the language for the predicates. In Prolog, each rule (or clause) is expressed as head :- body, where the ":-" operator means "if". Another feature of Prolog is that tokens beginning with a capitol letter are variables while tokens beginning with a small letter are constants. Thus, the **operational**() predicate states that a system state is considered to be operational if C, the number of operational processors, D, the number of operational disks and B, the number of operational network buses are all positive. Similarly, the **non-operational**() predicate says that the system is not operational if C, the number of operational processors, or B, the number of operational buses, or D, the number of operational disks, is 0. The **operational**() and **non-operational**() predicates can now be used in a pattern reward function. This can form the basis of a query to find for example, the distribution of cumulative operational time. For this query, a pattern expression consisting of two pattern states is necessary. A reward rate of one will be assigned to any state that satisfies the predicate **operational**() and a reward rate of zero will be assigned to any state that satisfies the predicate **non-operational**. The sequence reward function is as follows:

cumulative-operational((S:1 + T)*) where
operational(S),
non-operational(T).

For this sequence reward functions, the simpler representation as a pattern reward function can be used which does not explicitly specify the states with zero reward. Thus the above sequence reward function can be rewritten as:

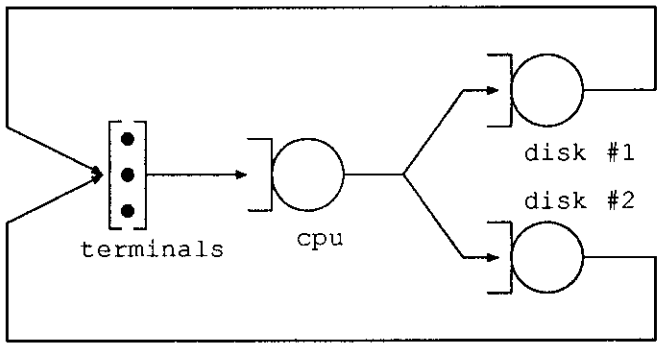

Figure 3: Computer Performance Model

cumulative-operational(S) yields 1 where
operational(S).

Besides the simpler pattern expression, the reward rate is specified as 1 in the "yields 1" portion of this clause[3]. The simpler pattern reward function suffices for many queries and will be used throughout the rest of the paper except where noted. The more complex form is sometimes required as will be illustrated by several examples in section 2.2.

Using the same system model, the number of system failures rather than the cumulative operational time might be desired. A system failure is a subsequence of two states, the first of which satisfies the **operational**() predicate and the second of which does not. An (impulse) reward of one will be assigned to each such subsequence. This type of reward function is typical of queries such as the distribution of the number of failures in an interval. The pattern reward function for this is as follows:

count-failures(S,T) yields 1 where
operational(S),
non-operational(T).

The last example in this section is a pattern reward function for throughput in a computer performance model. The model is of a simple time sharing system, consisting of some terminals, one processor, and two disks. There will be a number of jobs circulating among the different resources acquiring service. A service completion will be considered to be a departure of a customer from one of the disks. The state of the system will be represented as a four-tuple consisting the number of customers at the terminals, the number of customers at the cpu, the number of customers at the first disk, and the number of customers at the second

[3]Section 3 gives the formal description of how this clause is transformed into a deterministic finite state automata with output.

disk. Thus a system with two customers at the terminals, and one customer waiting at the second disk would be represented as $[[term, 2], [cpu, 0], [disk1, 0], [disk2, 1]]$. This model is shown graphically in Figure 3.

completion([[term,T1],[cpu,C],[disk1.D1],[disk2,D]], [[term,T2][cpu,C],{disk1,D2],[disk2,D]]) :-
D2 = D1 − 1,
T2 = T1 + 1.
completion([[term,T1],[cpu,C],[disk1,D],[disk2,D1]], [[term,T2][cpu,C],{disk1,D],[disk2,D2]]) :-
D2 = D1 − 1,
T2 = T1 + 1.

throughput(S,T) yields 1 where
completion(S , T).

Note that there are two clauses in the **completion**() pattern reward function. The first clause says that a completion occurs when a customer departs $disk1$ and arrives at the *terminals*. The second clause says that a completion also occurs when a customer departs $disk2$ and arrives at the *terminals*. Note also that the predicate **completion**() takes two arguments and is satisfied only when the two states are related to each other in a specific way. This is a much more difficult implementation problem than the other two queries, because the **completion**() predicate induces many different classes of states, roughly one for each pair of related states that satisfy the predicate. A sequence of states such as $[[term, 0], [cpu, 0], [disk1, 0], [disk2, 2]] \rightarrow [[term, 1], [cpu, 0], [disk1, 0], [disk2, 1]] \rightarrow [[term, 2], [cpu, 0], [disk1, 0], [disk2, 0]]$ for example has two service completions in succession. Thus care must be taken to recognize that the state $[[term, 1], [cpu, 0], [disk1, 0], [disk2, 1]]$ not only ends the first service completion, but it also begins the second.

2.2 Complete Queries

Now the pattern reward functions can be used in complete queries including the specific analysis measure desired. We give several examples of the types of queries that can easily be accommodated by our system. The first one continues the example computer reliability model with repair used in the previous section and shown in Figure 2.

The state of the system is described as a three-tuple, $[[cpu, C/C_0], [bus, B/B_0], [disk, D/D_0]]$ where C stands for the number of operational processing elements out of C_0 total, D stands for the number of operational disks out of D_0 total, and B stands for the number of operational buses out of B_0 (zero or one) total. The initial state of the system is represented as $[[cpu, 3/3], [bus, 1/1], [disk, 2/2]]$ meaning that all three processors are operational, both disks

are operational, and the bus is operational. The state of the system will evolve according to the sequence of component failures and repairs.

Two queries will be applied to this model. The first query is to find the (numerical) distribution of cumulative operational time $\mathcal{O}(s,t)$, over the interval $[0,t]$ which is $P[\hat{\mathcal{R}}(t) \leq s]$ where $s \leq t$. Expressing the query is quite simple.

```
1.  Q_1(S) yields 1 where
2.      operational(S).
3.
4.  operational([[cpu,C/C_0],[bus,B/B_0],[disk,D/D_0]]) :-
5.      C > 0,
6.      D > 0,
7.      B > 0.
8.
9.  d_cumulative_time(Q_1,10.9,1e-6, [[1.0,[[cpu,3/3],[bus,1/1],[disk,2/2]]]]).
```

Lines 1 and 2 define the pattern reward function for this query. In this case, all operational states regardless of context are assigned a reward of one. Lines 4 through 7 give the definition of an operational state, in this case, a state with at least one processor, one disk, and the network all operational. Finally, line 9 is the actual query. The exact result desired is the probability that the system is operational for at most 9 units of time in an interval of length 10. This type of query is listed in the table in Figure 1 as "d_cumulative_time" and requires 5 arguments, a pattern reward function, a time horizon, a threshold, an error bound, and an initial state distribution. The first argument of the query is the name of the pattern reward function, **Q_1** in this case. The next two arguments are the time horizon $t = 10$ and the threshold value $s = 9$. The fourth argument, 1e-6, is the error bound for the calculations, and the final argument in the initial state distribution where with probability 1.0 the system starts in the state where all components are operational.

A second example is the calculation of the distribution of the number of failures in an interval $[0,t]$. Specifically, a reward of one is generated if an operational state is followed in sequence by a non-operational state. This query can be expressed as follows.

```
1.  Q_2(S_1,S_2) yields 1 where
2.      operational(S_1),
3.      non-operational(S_2).
4.
5.  non-operational([[cpu,0/C_0],[bus,B/B_0],[disk,D/D_0]]).
6.  non-operational([[cpu,C/C_0],[bus,0/B_0],[disk,D/D_0]]).
```

```
7.  non-operational([[cpu,C/C_0],[bus,B/B_0],[disk,0/D_0]]).
8.
9.  d_number_events(Q_2,10,2,1e-6, [[1.0,[[cpu,3/3],[bus,1/1],[disk,2/2]]]]).
```

In this example, line 1 defines a pattern consisting of two states represented by S_1 and S_2. Lines 2 and 3 give the conditions on those states for the assignment of the reward, i.e. state S_1 must be operational and state S_2 must not be operational. Since the predicate **operational**() was defined in the previous example, only the predicate **non-operational()** has to be defined. The **non-operational()** predicate could be defined more simply as the logical complement of the **operational** predicate, but it is clearer to give the explicit definition. Lines 5, 6, and 7 define the three different cases of the predicate **non-operational**(). Line 5 defines the case where the number of operational processing elements is zero, line 6 defines the case where the number of operational buses is zero, and line 7 defines the case where the number of operational disks is zero. Finally line 9 calls the numerical routine to calculate the distribution of the number of events. In this specific call, the probability that there are 2 or less failures during the interval $[0, 10]$ is required with an error bound of 1e-6. Pattern reward function **Q_2** defines the events, 10 is the time horizon, 2 is the threshold on the number of events, 1e-6 is the error bound, and the initial state distribution is the same as before.

The next example is the performance model from the previous section and shown in Figure 3.

```
1.  Q_3(S_1,S_2) yields 1 where
2.      completion(S_1, S_2).
3.
4.  completion([[term,T1],[cpu,C],[disk1,D1],[disk2,D]], [[term,T2][cpu,C],[disk1,D2],[disk2,D]]) :-
5.      D2 = D1 − 1,
6.      T2 = T1 + 1.
7.  completion([[term,T1],[cpu,C],[disk1,D],[disk2,D1]], [[term,T2][cpu,C],[disk1,D],[disk2,D2]]) :-
8.      D2 = D1 − 1,
9.      T2 = T1 + 1.
10.
11. e_number_events(Q_3,100,1e-6, [[1.0,[[term,3],[cpu,0],[disk1,0],[disk2,0]]]]).
```

The pattern reward function, **Q_3**, defines a pattern on two states, S_1 and S_2. However, unlike the last query where the states were not directly related, in this query, S_1 and S_2 are directly related by the predicate **completion**(). Lines 4 through 9 are a (repeat) definition of a service completion. Line 11 calls the numerical calculation for the expected number of events which with the arguments **Q_3**, the pattern reward function, 100, the time interval,

and 1e-6, the error bound, and finally the initial state distribution with a total of three customers all at the terminals with probability 1.0.

The last example is a simple performability model that uses a sequence reward function. In this system there are two processors that fail and can be repaired. Customers arrive at some rate and enter a queue for service. Depending on how many processors are operational, zero, one, or two of the customers in the queue can be receiving service. The state for this system can be of the form $[cpu, Nq, Up/Total]$ where Nq is the number of customers in the queue, Up is the number of operational processors, and $Total$ is the total number of processors. Thus a completely operational system with two customers in the queue and receiving service would be represented as $[cpu, 2, 2/2]$. The measure desired from this system is the distribution of the total service supplied to customers before both processors fail.

```
1.  Q_4( (S_0 + S_1:1 + S_2:2)* S_3 S_4* ) where
2.       ratezero(S_0),
3.       rateone(S_1),
4.       ratetwo(S_2),
5.       failed(S_3),
6.       any(S_4).
7.
8.  ratezero([cpu,0,X/Y]) :- X > 0.
9.  rateone([cpu,1,X/Y]) :- X > 0.
10. ratetwo([cpu,N,X/Y]) :- N > 1, X > 1.
11. failed([cpu,W,0/Y]).
12. any(X).
13.
14. d_total_reward(Q_4,10,12,1e-6,[[1.0,[cpu,0,2/2]]]).
```

To express this measure, we need a sequence expression with four pattern states as shown in lines 1-6. The first part of the sequence expression, $(S\_0 + S\_1 : 1 + S\_2 : 2)*$, allows any initial subsequence of Markov chain states with one or two processors operating. During this period, reward is accumulated according to how many of the processors are providing service. The remainder of the sequence expression, $S\_3\ S\_4*$ allows any sequence of states after a failed state but provides no more reward. Lines 8-12 give the predicates for this query. The predicate, **ratezero()**, in line 8 is satisfied by any state with 0 customers in the queue but at least one operational processor. Lines 9-11 are similar and need no further explanation. Predicate **any()** in line 12 is satisfied by all states. Line 14 provides the call to the actual numerical solution routine.

The query language only provides the means for the user to express what is desired. Each query then has to be translated to a form that can be efficiently executed. The next section

shows how to translate pattern reward functions into a type of finite automata with output. Such finite state automata are called Mealy machines [HOPC79]. The Mealy machine is then used by different numerical analysis routines to compute the final query results.

3 Translation to a Mealy Machine

In this section it is shown how to convert a pattern reward function into a type of finite state automata called a Mealy machine. Translating a sequence reward function is similar and is omitted for conciseness. Once the translation is accomplished, one effectively has an implementation of the query language since numerical results can be computed given a Mealy machine. The simpler problem of converting a single clause of a pattern reward function into a Mealy machine is first examined. After that, the general problem of converting pattern reward functions consisting of several clauses to a Mealy machine is studied.

With each transition of the Markov chain, an output is generated as a function of the sequence of states leading up to and including the current state. It is shown that this output function can be represented by a type of finite automata with output. Each state of the automata will correspond to an equivalence class of initial sequences of states of the Markov chain. This type of abstract machine has been described in the literature (see [HOPC79] for example) and is called a *Mealy machine*. A Mealy machine is a six-tuple $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where Q is a set of states, Σ is the input alphabet (i.e. Markov chain states), q_0 is the initial automata state, and δ is a function which maps $Q \times \Sigma$ into Q. These attributes are identical to those in a deterministic finite automata. In addition, Δ is a set of machine outputs and λ maps $Q \times \Sigma$ to Δ, that is $\lambda(q, s) \forall q \in Q, \forall s \in \Sigma$ gives the output associated with the transition from state q on input s. We define $\hat{\delta}(s_1 s_2 \cdots s_n)$, the multistep transition function to be equal to $\delta(\delta(\cdots(\delta(s_1), s_2), \cdots), s_n)$. For simplicity, assume that the rewards Δ are all integers and define $\vec{e}_i$ to be the vector of all 0's except for a 1 in the i^{th} position. Similar to $\hat{\delta}$, $\hat{\lambda}(s_1 s_2 \cdots s_n)$, the multistep *output* of machine M in response to input $s_1 s_2 \cdots s_n$ is defined to be $\vec{e}_{\lambda(q_0, s_1)} + \vec{e}_{\lambda(q_1, s_2)} + \cdots \vec{e}_{\lambda(q_{n-1}, s_n)}$ where $q_0, q_1, \cdots, q_n$ is the sequence of states such that $\delta(q_{i-1}, s_i) = q_i$ for $1 \leq i \leq n$. The *reward* from an output sequence is just the vector containing the number of times each different output value occurs in the output sequence. Thus the output sequence 1, 2, 1, 3 would have a reward vector of $[2, 1, 1]$ associated with the reward levels $[1, 2, 3]$. In the special case where there is one distinct non-zero reward, the vector has only one component. Other simplifications are possible depending on the type of numerical analysis or type of measure and these variations will be discussed in section 4.

As a simple example of a Mealy machine, consider the reliability model from the previous section. One pattern reward function given for this model was

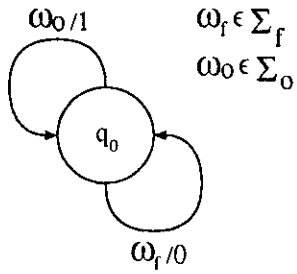

Figure 4: Mealy machine for identifying operational states

cumulative-operational(S) yields 1 where
operational(S).

Let Σ_o be the set of Markov chain states that satisfy the **operational**() predicate, and Σ_f be the set of Markov chain states that do not satisfy the **operational**() predicate. The Mealy machine M for this reliability example would consist of only one state as the output does not depend on the history at all. The entire machine would be $M = (\{q_0\}, \Sigma_f \cup \Sigma_o, \{0,1\}, \delta, \lambda, q_0)$ where $\delta(q_0, \omega_o) = \delta(q_0, \omega_f) = q_0$ for all $\omega_o \in \Sigma_o$ and $\omega_f \in \Sigma_f$ and where $\lambda(q_0, \omega_o) = 1$ and $\lambda(q_0, \omega_f) = 0$.

It is usually easier to understand Mealy machines represented graphically than in terms of symbols. The graphical notation is similar to that used for finite automata. Each (machine) state $q \in Q$ is represented by an ellipse (usually a circle) and each transition is represented by an arrow from one state to another. Each arrow is labeled by the Markov chain state $\sigma \in \Sigma$ that causes the transition plus the reward accumulated because of that transition. Thus, a transition out of a state q_0 labeled $\omega_f/1$ would output a reward of one if a Markov chain state $\sigma \in \Sigma_f$ occurred while in machine state q_0. A transition labeled with just a Markov chain state (i.e. no reward designation) is assumed to have a reward of zero. The pattern reward function for computing the cumulative operational time is shown graphically in Figure 4.

A slightly more complex reliability example is to count the number of failures. A failure is a transition from a state which is up (operational) to a state which is down. In this case, the reward is not determined completely by the current (Markov chain) state, but rather the current state and the immediately preceding state. The pattern reward function for this example was given in the previous section as

count-failures(S,T) yields 1 where
operational(S),
not-operational(T).

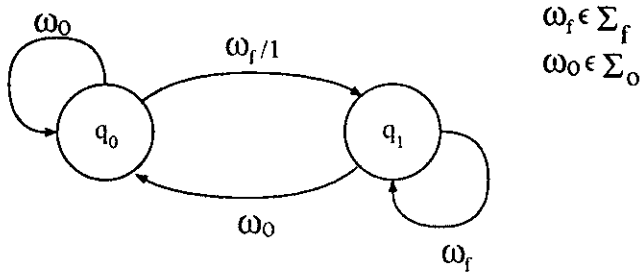

Figure 5: Mealy machine for counting the number of failures

Two automata states are needed for this machine. State q_0 will represent the case where the last Markov chain state was up, and state q_1 will represent the case where the last Markov chain state was down. The system being modeled has the same Markov chain states as before, $\Sigma_f \cup \Sigma_o$. The Mealy machine for this query will be $(\{q_0, q_1\}, \Sigma_f \cup \Sigma_o, \{0,1\}, \delta, \lambda, q_0)$ where $\delta(q_0, \omega_o) = q_0, \delta(q_0, \omega_f) = q_1, \delta(q_1, \omega_o) = q_0, \delta(q_1, \omega_f) = q_1$. The output function λ for this query is $\lambda(\omega_o, q_0) = 0, \lambda(\omega_f, q_0) = 1, \lambda(\omega_o, q_1) = 0, \lambda(\omega_f, q_1) = 0$. It is easy to see that the reward for this query with the input, $[\sigma_1, \sigma_2, \sigma_3]$ with $\sigma_1 \in \Sigma_o$ and $\sigma_2, \sigma_3 \in \Sigma_f$ is equal to 1 with this query whereas it was equal to 2 with the previous one. The graphical representation of this query is shown in Figure 5.

In the remainder of this section we present a detailed description of the translation of pattern reward functions to Mealy machines including the case in which there are several pattern reward function clauses.

The first step in building a Mealy machine from a pattern reward function clause is to create a *nondeterministic Mealy machine* from each clause of the pattern reward function. A nondeterministic Mealy machine is a Mealy machine that allows ϵ-transitions (a transition with no input) and also allows two different transitions on the same input. To translate a clause of a pattern reward function to a nondeterministic Mealy machine, treat the pattern expression as a nondeterministic finite automata with one extra transition. This extra transition goes from the initial state of the automata back to itself and can occur on any Markov chain state. Consider the pattern reward clause **recover**()

recover(S,T,U) yields 1 where
operational(S),
non-operational(T),
operational(U).

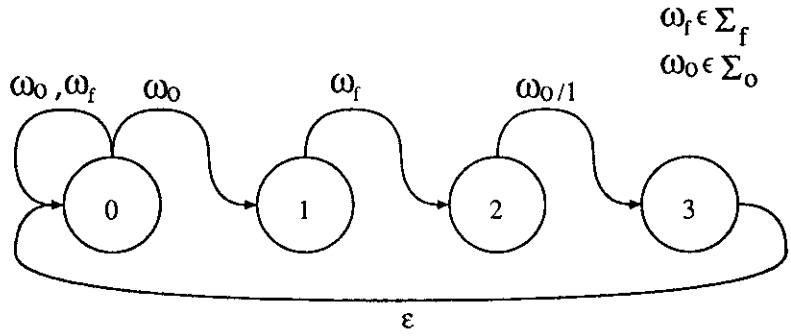

Figure 6: Correct nondeterministic machine

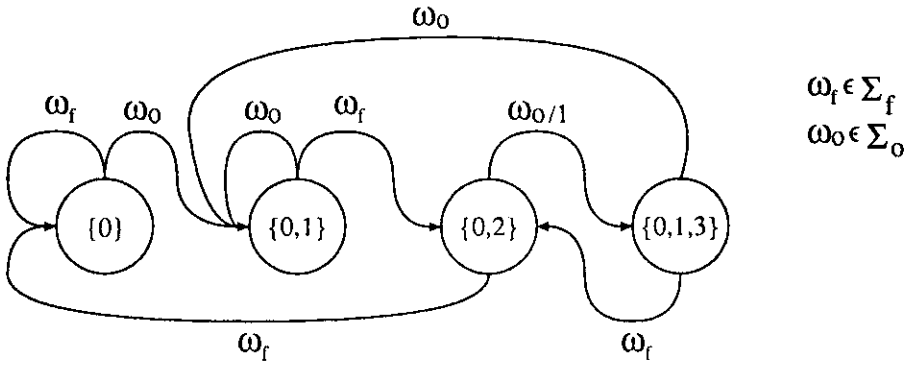

Figure 7: Correct deterministic machine

This clause can be translated to the nondeterministic machine shown in Figure 6. The machine illustrated in Figure 6 can be converted to an equivalent deterministic one by extending known algorithms [AHO74, HOPC79] to deal with the machine output. The basic construction is to deal with sets of states rather than individual states. Thus there would be a deterministic transition from state $\{0\}$ to state $\{0,1\}$ on entering a Markov chain state ω_o since there is a transition from automata state 0 back to itself and from automata state 0 to automata state 1 when the input is ω_o. The complete deterministic Mealy machine for this pattern reward clause is shown in Figure 7.

3.1 Translating multiple clauses

We now consider how to compile two pattern reward function clauses, each represented by a Mealy machine, into one machine. This process can be repeatedly applied to combine three or more machines together. We represent a state in the cross-product machine as a 2-tuple, (q_i^1, q_j^2) where q_i^1 is the state q_i from the first machine and q_j^2 is the state q_j from the second

machine. Each transition in this machine is applied to both the elements of the tuple. If the rewards from the two machines are the same, that value is the combined reward. If one of the rewards is zero, the non-zero value is used. If the rewards are distinct and nonzero, then the two clauses must be inconsistent. More formally, let $M_1 = (Q_1, \Sigma, \Delta_1, \delta_1, \lambda_1, q_0^1)$ and let $M_2 = (Q_2, \Sigma, \Delta_2, \delta_2, \lambda_2, q_0^2)$. We will define $M = M_1 \times M_2 = (Q_1 \times Q_2, \Sigma, \Delta_1 \cup \Delta_2, \delta, \lambda, (q_0^1, q_0^2))$ where $\delta((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$ and where $q_1 \in Q_1, q_2 \in Q_2$, and $\sigma \in \Sigma$. Since the reward functions λ_i for the individual machines for the same sequence are either supposed to be the same or at least one of them must be zero, the reward function $\lambda(\sigma, (q_1, q_2)) = \max(\lambda_1(\sigma, q_1), \lambda_2(\sigma, q_2))$. The new machine M, in effect, steps through both machines simultaneously. Any inconsistencies in the query will be manifest at this point as two or more of the rewards having unique nonzero values, which is easily detected and signaled to the user.

As an example of this construction, consider again a reliability model with three states, operational, degraded, and non-operational (with, for simplicity, no transitions from the operational state directly to the non-operational state). One pattern reward function on this model may consist of the following two clauses.

fail(S,T) yields 1 where
    operational(S),
    degraded(T).
fail(S,T) yields 1 where
    degraded(S),
    non-operational(T).

This pattern reward function counts the sum of transitions from operational to degraded mode and from degraded mode to non-operational. The Mealy machine corresponding to the combination of these two clauses is shown in Figure 8 where u (for up) is used to represent an operational Markov chain state, d is used to represent a degraded Markov chain state, and D (for down) is used to represent a non-operational Markov chain state. Note that the example sequence of states *operational* $\rightarrow$ *degraded* $\rightarrow$ *nonoperational* would satisfy both clauses of the **fail**() reward function since the transition *operational* $\rightarrow$ *degraded* satisfies the first clause, while the transition *degraded* $\rightarrow$ *nonoperational* satisfies the second. This sequence is assigned the correct reward by the combined machine. To make this clearer, the path for this example sequence is highlighted in Figure 8. It should also be clear that $\times$ is associative, so that $(M_1 \times M_2) \times M_3 = M_1 \times (M_2 \times M_3) = M_1 \times M_2 \times M_3$, and states can be represented as (q_0^1, q_0^2, q_0^3) rather than $((q_0^1, q_0^2), q_0^3)$ or $(q_0^1, (q_0^2, q_0^3))$.

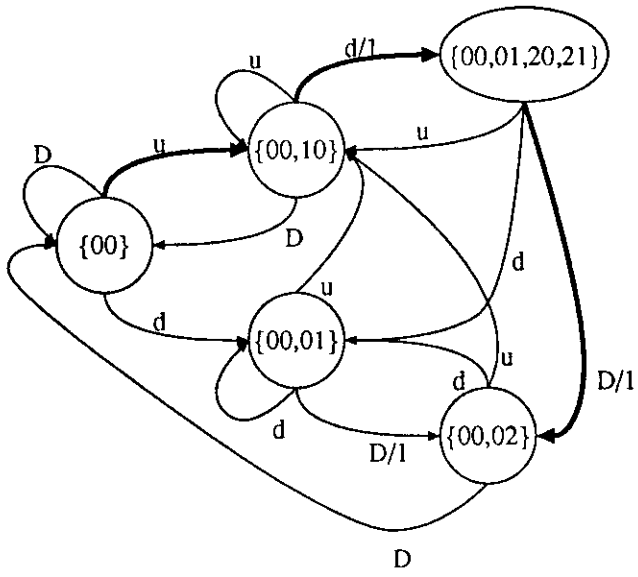

Figure 8: Correct combined deterministic machine

3.2 Minimization

We would like to be able to minimize this machine for fast execution and to minimize storage requirements. To do this, we extend the usual algorithms [HARR65] for minimizing finite automata to deal with the machine output as follows. We define two states p and q to be *equivalent* with respect to machine M, written $p \overset{M}{\equiv} q$, if the output from machine M started in state p is the same as the output from machine M started in state q for any string of input. We will write this as $p \equiv q$ when there will be no confusion as to the machine. The operator $\equiv$ is an equivalence relation and defines a partition over the set of states. Each equivalence class forms a state in the new minimal machine. Thus by finding the equivalence classes, the minimal machine is found. A state p is distinguishable from a state q if states p and q are not equivalent, that is if there exists an input sequence x such that $\hat{\lambda}(p,x) \neq \hat{\lambda}(q,x)$. The algorithm is initialized by marking all pairs of states that are distinguishable by a sequence of length one as distinguishable and all the other states as unknown, that is if $\lambda(p,a) \neq \lambda(q,a)$ for some a, then states p and q are distinguishable. Next for each pair of states r and s that are not already known to be distinguishable, the pairs of states $t = \delta(r,a)$ and $u = \delta(s,a)$ for each input symbol a are considered. If states t and u have been shown to be distinguishable by some string x, then r and s can be distinguished by some string ax. If states t and u have not been shown to be distinguishable, then the pair (r,s) is placed in a list associated with (t,u). At some future time, if the (t,u) entry is found to be distinguishable, then each pair in the list associated with (t,u) is marked as distinguishable. This process is repeated until a pass is made through all the pairs of states without finding any new pairs to be distinguishable. At this point, the algorithm is complete and the new states are the sets of

```
1.  initialize unknown[ ], distinguishable[ ]
2.  done := false
3.  while not done begin
4.      done := true
5.      for each pair (r, s) ∈ distinguishable[ ]
6.          for each a ∈ Σ begin
7.              let t = δ(r, a), u = δ(s, a)
8.              if (t, u) ∈ distinguishable[ ] begin
9.                  done := false
10.                 move (r, s) to distinguishable[ ] from unknown[ ]
11.                 for each pair linked to (r, s) move to distinguishable[ ]
12.             end else
13.                 link (r, s) to (t, u)
14.         end
15. end
```

Figure 9: Algorithm for minimizing automata

indistinguishable states. This algorithm is summarized in Figure 9.

4 Markov Model Solution

This section describes the calculation of the response to a query. Starting with a description of a continuous time Markov chain $X(t)$ and a query, it is shown how to evaluate that query on the given Markov chain description. The first step is to generate the Markov chain transition rate matrix as is described for example in [BERS90]. For our purposes here, the only important point is that this procedure creates a list of transition rates in the form of triples, (S_i, S_j, R_{ij}), each of which represents a transition from state S_i to state S_j that occurs with a rate of R_{ij}. It is useful to be able to normalize this transition rate matrix into a transition probability matrix where the time between any two transitions is exponentially distributed with the same mean. This can be done by a technique called randomization (or uniformization) [GROS84]. If $Q = \{q_{ij}\}$ is the transition rate matrix and $\Lambda \geq \sup_i \mid q_{ii} \mid$, then results for the matrix Q can be obtained from the transition probability matrix $P = Q/\Lambda + I$. In the randomized Markov chain, each state has the same exponential holding time distribution. For measures over an interval of time $(0, t)$, the solution is obtained by conditioning and then unconditioning over the number of (Poisson distributed) transitions that occur in $(0, t)$. Thus the reward from a query represented by a function $f(\cdot)$

$$F(t) = \int_0^t f(X(u))du$$

can be rewritten as

$$F(t) = \sum_{n=0}^{\infty} \frac{e^{-\Lambda t}(\Lambda t)^n}{n!} f_n(\pi(0)P^n)$$

All of the measures described in this paper can be expressed by the proper choice of the series of functions $f_n(\cdot)$. For example, if it is desired to compute $F(t) = \pi(t)$, the time dependent state probability vector, then set $f_n(x) = x$. Note that as long as $f_n(\cdot)$ is bounded above by $\bar{f}$ for $n > N$,

$$\begin{aligned} F(t) &= \sum_{n=0}^{N} \frac{e^{-\Lambda t}(\Lambda t)^n}{n!} f_n(\pi(0)P^n) + \sum_{n=N+1}^{\infty} \frac{e^{-\Lambda t}(\Lambda t)^n}{n!} f_n(\pi(0)P^n) \\ &\leq \sum_{n=0}^{N} \frac{e^{-\Lambda t}(\Lambda t)^n}{n!} f_n(\pi(0)P^n) + \bar{f} \sum_{n=N+1}^{\infty} \frac{e^{-\Lambda t}(\Lambda t)^n}{n!} \\ &\leq \sum_{n=0}^{N} \frac{e^{-\Lambda t}(\Lambda t)^n}{n!} f_n(\pi(0)P^n) + \bar{f}\left[1 - \sum_{n=0}^{N} \frac{e^{-\Lambda t}(\Lambda t)^n}{n!}\right] \end{aligned}$$

Thus the error due to truncating the infinite series can be bounded (if the $f_n(\cdot)$ can be bounded) and by choosing N large enough, the error term can be made arbitrarily small. Before describing the functions, $f_n(\cdot)$, a new Markov chain derived from P must be defined which is a function of P and also the query to be evaluated.

A new Markov chain is produced by incorporating the reward level and the state of the finite automata (derived from the pattern reward function as described in the previous section) into the transition probability matrix P. The finite state automata will be represented by the two functions, δ, the automata transition function, and λ, the automata reward function. The original Markov chain, the automata transition function, and the accumulated reward vector[4] are combined into a new Markov chain with states $(A, S, \vec{K})$ where A is an automata state, S is a Markov chain state, and $\vec{K}$ is a reward vector. Since the holding time distributions of all states in the randomized Markov chain are the same, the reward vector $\vec{K}$ can be interpreted as giving the number of occurrences of each different reward rate. Let $p(S_i, S_j)$ be the transition probabilities in the original (randomized) chain and let $\vec{e_i}$ be a vector of all 0's except for a 1 in the i^{th} position, then define P' to be the transition probability matrix having the following structure

$$p'[(A, S, \vec{K}), (A', S', \vec{K}')] = \begin{cases} p[S, S'] & \text{if } \delta(A, S') = A' \text{ and } \vec{K}' = \vec{K} + \vec{e}_{\lambda(A', S')} \\ 0 & \text{otherwise} \end{cases}$$

[4]For simplicity, it is assumed that rewards are restricted to non-negative integers

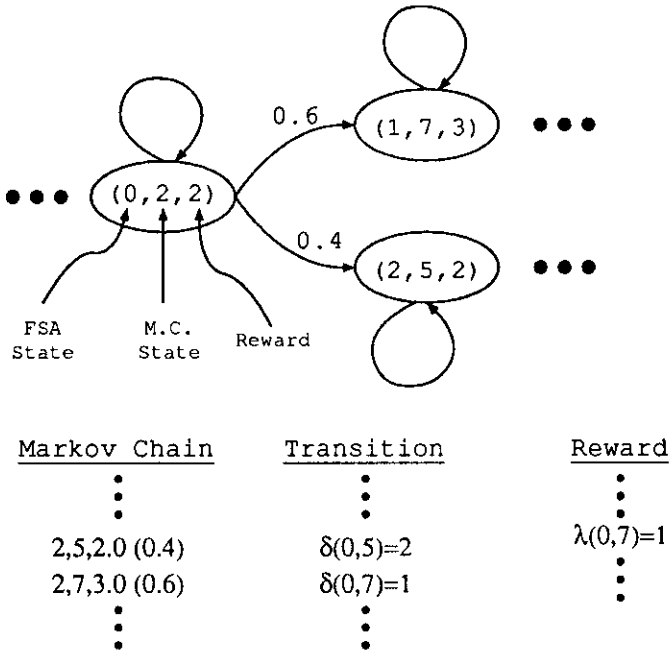

Figure 10: Conceptual Markov chain with automata state and rewards

In general, we need to compute the probabilities that a certain level of reward is accumulated by the sample paths of this Markov chain. This will be collected iteratively in a vector of three dimensions $X^n(A, S, \vec{K})$ where A is the automata state, S is the Markov chain state, $\vec{K}$ is the vector of accumulated rewards, and n is the iteration number. We can now iteratively define

$$\begin{aligned} X^{n+1}(A, S, \vec{K}) &= X^n(A, S, \vec{K})P' \\ X^0(A_0, S_0, \vec{e}_{\lambda(A_0,S_0)}) &= 1 \\ X^0(A, S, \vec{K}) &= 0 \text{ for all other } A, S \text{ and } \vec{K} \end{aligned}$$

where A_0 and S_0 are the initial automata state and the initial Markov chain state respectively and $\lambda(A_0, S_0)$ is the Mealy machine initial output.

These relationships are shown graphically in Figure 10. Figure 10 shows only part of the Markov chain but should serve to illustrate the concept. Each state in the Markov chain is listed as a triple (A, S, K) where S is the current Markov chain state, A is the current automata state, and K is the current reward vector. For simplicity, the rewards in this model are restricted to 0 or 1 and so the reward vector K is shown as a scalar value for the number of 1's. The column titled *Markov Chain* gives the Markov chain transitions and their rates. The number in parentheses is the (randomized) transition probability. Thus, the first entry in the table

2,5,2.0 (0.4)

represents a transition from Markov chain state 2 to state 5 at a rate of 2.0, or equivalently a probability of 0.4. The column labeled *Transition* represents δ, the automata transition function. Finally, the column labeled *Reward* contains the representation of λ, the reward function. To find a transition from the combined Markov chain state (0,2,2), the Markov chain transition table is searched for transitions out of Markov chain state 2. Two transitions are found, one to Markov chain state 5 at rate 2.0 (probability 0.4) and one to state 7 at rate 3.0 (probability 0.6). The automata transition has to be accounted for in each of these two transitions. In the case of the transition to Markov chain state 7 while in automata state 0, the automata transition table shows the next automata state is 1. Finally the reward is given in the reward table as 1 for a transition to Markov chain state 7 while in automata state 0. Thus there is a transition from combined state (0,2,2) to (1,7,3) where the reward 3 is just the old reward plus the reward output of 1. Other transitions are similar.

Now we describe how the $f_n(\cdot)$'s are determined. For the following examples assume that there are $K+1$ different levels of reward that can be earned. Let $\vec{r} = (r_1, r_2, \ldots, r_{K+1})$ be the vector containing these reward levels such that $r_1 > \cdots > r_K > r_{K+1} \geq 0$. Let $\vec{k} = (k_1, k_2, \ldots, k_{K+1})$ be the vector with k_i equal to the number of visits to a state with reward rate r_i. Recall that $X^n(A, S, \vec{K})$ is a vector computed over sequences of Markov chain states of length $n+1$ and that due to randomization, the time spent in each state of the sequence has the same distribution. Therefore, in an interval of time $(0, t)$ with n transitions (and $n+1$ Markov chain states), the expected time in each state is $t/(n+1)$. Multiplying the reward for each state by the time in each state and by the probability of each state gives the $f_n(\cdot)$'s to compute the expected reward.

$$f_n(X^n(A, S, \vec{K})) = \sum_{a,s,\vec{k}} (\vec{k} \cdot \vec{r}) \frac{t}{n+1} X^n(a, s, \vec{k})$$

The second example illustrates the functions to compute the distribution of total reward, $\mathcal{Y}(R, t) = P[\mathcal{R}(t) \leq R]$. The derivation of the functions $f_n(\cdot)$ for this measure is included for completeness; for additional details, see [DESO89]. Let $\zeta = (\zeta_1, \ldots, \zeta_{K+1})$ where ζ_i is the sum of the lengths of intervals of reward r_i. Then the total reward earned is $\mathcal{R}(t) = \vec{r} \cdot \vec{\zeta}$. For a Markov chain state sequence of n transitions (i.e. $n+1$ states), let the intervals of time spent in each state be denoted by $Y_i, 1 \leq i \leq n+1$. Due to the fact that these Y_i's are exchangeable [DESO89], we can assume that the first k_i intervals are of reward r_i. That is,

$$\begin{aligned} \zeta_1 &= Y_1 + \cdots + Y_{k_1} \\ \zeta_2 &= Y_{k_1+1} + \cdots + Y_{k_1+k_2} \\ &\vdots \end{aligned}$$

$$\begin{aligned}\zeta_K &= Y_{k_1+\cdots+k_{K-1}+1} + \cdots + Y_{k_1+\cdots+k_K} \\ \zeta_{K+1} &= Y_{k_1+\cdots+k_K+1} + \cdots + Y_{n+1}\end{aligned}$$

Let $U_1, \ldots, U_{n+1}$ be independent and identically distributed random variables uniform on $(0,t)$ and let $U_{(k)}, 1 \leq k \leq n+1$ be the kth order statistic (i.e. $U_{(k)}$ is the kth smallest of the U_i). The above equations can be rewritten

$$\begin{aligned}\zeta_1 &= U_{(k_1)} \\ \zeta_2 &= U_{(k_1+k_2)} - U_{(k_1)} \\ &\vdots \\ \zeta_K &= U_{(k_1+\cdots+k_K)} - U_{(k_1+\cdots+k_{K-1})} \\ \zeta_{K+1} &= t - U(k_1 + \cdots + k_K)\end{aligned}$$

Defining $n_j = \sum_{l=1}^{j} k_l$ for $1 \leq j \leq K$, we may write

$$\mathcal{R}(t|n, \vec{k}) = \sum_{j=1}^{K} (r_j - r_{j+1}) U_{(n_j)} + r_{K+1} t$$

where $\mathcal{R}(t|n, \vec{k})$ is the reward accumulated in the interval $(0,t)$ given n and $\vec{k}$. We can also write

$$P\left[\mathcal{R}(t) \leq R|n, \vec{k}\right] = P\left[\sum_{j=1}^{K} (r_j - r_{j+1}) U_{(n_j)} \leq R - r_{K+1} t\right]$$

In order to determine this probability, we need to find the distribution of a linear combination of order statistics from a uniform distribution on $(0,t)$. Now define

$$\bar{s} = R - r_{K+1} t$$

and, for $1 \leq i \leq K$, define

$$s_i = r_i t - r_{K+1} t$$

Let

$$m = \text{the largest index } i \text{ such that } R \leq r_i t$$

Then a theorem of Weisberg [DAVI81] says that

$$P\left[\sum_{j=1}^{K}(r_j - r_{j+1})U_{(n_j)} \leq \bar{s}\right] = 1 - \sum_{i=1}^{m}\frac{g_i^{(k_i-1)}(s_i)}{(k_i-1)!}$$

where $g_i^{(k)}(x)$ is the k^{th} derivative of the function

$$g_i(x) = \frac{(x-\bar{s})^n}{\prod_{\substack{l=1\\ l\neq i}}^{K+1}(x-s_l)^{k_l}}$$

Note that there is a small problem when any of the k_i are zero. To deal with this, let $l(\vec{k},i)$ be the index of the i^{th} nonzero k_i and let $m(\vec{k})$ be the largest index i such that $R \leq r_{l(\vec{k},i)}t$. Then

$$f_n(X^n(A,S,\vec{K})) = \sum_{a,s,\vec{k}} X^n(a,s,\vec{k})\left\{1 - \sum_{i=1}^{m(\vec{k})}\frac{g_{l(\vec{k},i)}^{(k_{l(\vec{k},i)}-1)}(s_{l(\vec{k},i)})}{(k_{l(\vec{k},i)}-1)!}\right\}$$

The last example is the distribution of cumulative time, which is a special case of the distribution of total reward in which the rewards r_i are restricted to either 0 or 1. This restriction allows the reward vector $\vec{k}$ to be represented as a scalar $\bar{k}$, the number of times a reward of 1 is accrued. This reduction of the vector $\vec{k}$ to the scalar $\bar{k}$ can be done whenever there are only two distinct reward rates. The functions (see [DESO86]) $f_n(\cdot)$, with R and t the threshold and the time horizon as before, are

$$f_n(X^n(A,S,\bar{k})) = \sum_{a,s,k} X^n(a,s,k)\sum_{i=k}^{n}\binom{n}{i}\left(\frac{R}{t}\right)^i\left(1-\frac{R}{t}\right)^{n-i}$$

Summary

For the numerical solution, the Markov chain triples, the automata transition function triples, and the automata reward function triples are sent to the backend solver along with (as needed) an initial state, a time horizon, an error bound and a threshold. The backend solver solves the query on the given Markov chains and returns the results. The pruning and aggregation techniques that are used by the backend solver to reduce computation and storage size requirements are described next.

4.1 Aggregating and Pruning

$X^n(A, S, \vec{k}), n = 0, 1, 2, \ldots$ represents results from possible sample path sequences. There are two approaches to optimization in processing these sequences: aggregating and pruning. Pruning consists of eliminating from current and future consideration certain sequences of states. Pruning provides savings not only for the sequences removed from consideration, but also for all possible continuations of these sequences. These savings can be quite large. Sequences are pruned usually for one of three reasons. The first is that the sequence has irrevocably failed to match the pattern. In this case, there is no reason to continue exploring sequences derived from this pattern. The second reason is that the sequence from this point on will always match the pattern. This occurs for example, in first passage time queries. After the first occurrence of the pattern of interest, it is unnecessary to generate alternative continuations of the sequence. The third and final type of pruning is applied when the probability of a sequence becomes insignificant. Since the probability of occurrence of a sequence is the product of the state transition probabilities that appears in the sequence, the sequence probability gets smaller as the sequence grows. If the probability of a sequence gets sufficiently small (e.g., relative to other sequences) as to not significantly affect the final query result, that sequence can be pruned. Criteria for determining when the probability of a sequence is sufficiently small are discussed later in this section.

Another approach to saving on calculations is aggregation. While pruning consists of eliminating certain sequences entirely, aggregating consists of combining equivalent sequences of states. By combining sequences of states, only one set of calculations, rather than two or more, need to be done from the point where the sequences are aggregated. This method can also save much computation. Both pruning and aggregating are performed when actually evaluating a query.

Aggregation of States

Aggregation consists of lumping together sequences that are equivalent. This has already been briefly described earlier in this section, as part of the definition of $X^n(A, S, \vec{k})$. Each entry $X^n(a, s, \vec{k})$ is the sum of the probabilities of many sequences. If $z^{(n)}$ is a sequence of n Markov chain states, $P[z^{(n)}]$ is its probability, and $\hat{\delta}(z^{(n)})$ is the Mealy machine multistep transition function, then

$$X^n(a, s, \vec{k}) = \sum_{z^{(n-1)}s} P[z^{(n-1)}s] \text{ where } \hat{\delta}(z^{(n-1)}s) = a \text{ and } \hat{\lambda}(z^{(n-1)}s) = \vec{k}$$

The level of aggregation in X^n above is appropriate for computing the distribution of total reward and also transient point probability.

In other types of measures such as expected total reward, expected cumulative time, expected first passage time, and expected number of events, the X^n's can be further aggregated. We look specifically at expected total reward, but the other cases are similar. Let $\{Y(t), t \geq 0\}$ be a Markov chain and let $\{X(t), t \geq 0\}$ be the Markov chain whose state space is the set of all (A, S) where A is an automata state and S is a state of the Markov chain $Y(t)$. Assuming that $X(t)$ can be randomized, let $\{N(t), t \geq 0\}$ be the counting process of the underlying Poisson process and let $\{T_n, n = 1, 2, \ldots\}$ be the times of occurrence of the Poisson process. Let $\lambda(\cdot)$ be the automata output function as earlier. Then following [GROS84]

$$\begin{aligned}
E\left[\int_0^t \lambda(X(u))du\right] &= \sum_{n=0}^{\infty} E\left[\int_0^t \lambda(X(u))du | N(t) = n)\right] P(N(t) = n) \\
&= \sum_{n=0}^{\infty} E\left[\sum_{j=0}^{n} \lambda(X(T_j))(T_{j+1} - T_j) | N(t) = n\right] P(N(t) = n) \\
&= \sum_{n=0}^{\infty} \sum_{j=0}^{n} E\left[\lambda(X(T_j))(T_{j+1} - T_j) | N(t) = n\right] P(N(t) = n) \\
&= \sum_{n=0}^{\infty} \sum_{j=0}^{n} E\left[\lambda(X(T_j))\right] E\left[T_{j+1} - T_j | N(t) = n\right] P(N(t) = n) \\
&= \sum_{n=0}^{\infty} \sum_{j=0}^{n} E\left[\lambda(X(T_j))\right] \frac{t}{n+1} P(N(t) = n) \\
&= \sum_{n=0}^{\infty} \frac{e^{-\Lambda t}(\Lambda t)^n}{n!} \frac{t}{n+1} \sum_{j=0}^{n} E\left[\lambda(X(T_j))\right]
\end{aligned}$$

Examine the term $\sum_{j=0}^{n} E[\lambda(X(T_j))]$

$$\sum_{j=0}^{n} E\left[\lambda(X(T_j))\right] = E\left[\sum_{j=0}^{n} \lambda(X(T_j))\right]$$

The term $\sum_{j=0}^{n} \lambda(X(T_j))$ is just the sum of the rewards at each step of the Markov chain which is $(\vec{r} \cdot \vec{k})$.

$$E\left[\sum_{j=0}^{n} \lambda(X(T_j))\right] = \sum_{a,s} \sum_{\vec{k}} (\vec{k} \cdot \vec{r}) X^n(a, s, \vec{k})$$

Finally we get

$$f_n(X^n(A, S, \vec{K})) = \frac{t}{n+1} \sum_{a,s} \sum_{\vec{k}} (\vec{k} \cdot \vec{r}) X^n(a, s, \vec{k})$$

From the structure of the functions $f_n(\cdot)$ it is easy to see that it is not necessary to store the distribution of $\vec{k}$. Since the rewards depend only on the weighted average of the reward vector $\vec{k}$. So only the weighted average need be stored instead of the whole distribution. Thus $X^n(a, s, \vec{k})$ can be stored in aggregated from as $X^n(a, s)$.

$$X^n(a,s) = \sum_{\vec{k}} (\vec{k} \cdot \vec{r}) X^n(a, s, \vec{k})$$

$X^n(a, s)$ can be iteratively calculated as the product of $X_p^n(a, s)$, the probability of being in automata state a and Markov chain state s at iteration n, and $X_r^n(a, s)$, the reward associated with a, s, and n

$$X_p^{n+1}(a', s') = \sum_{\substack{a,s \\ \delta(a,s')=a'}} X_p^n(a, s) p(s, s')$$

$$X_r^{n+1}(a', s') = \sum_{\substack{a,s \\ \delta(a,s')=a'}} \left[\lambda(a, s) + X_p^n(a, s) X_r^n(a, s) p(s, s')\right]$$

where $\delta(a, s)$ and $\lambda(a, s)$ are the Mealy machine transition function and output function, respectively and $p(s, s')$ is the uniformized probability of a Markov chain transition from state s to s'.

The last measure is steady state reward. In this case, the automata state a is unnecessary since the reward rates can be computed from the steady state probabilities. For example, if the throughput T, defined as the sequence of Markov chains states $s_1 s_2$, was desired from a model, it could be computed as

$$T = \pi_1 r[s_1, s_2]$$

where π is the vector of steady state probabilities, π_1 is the steady state probability of state 1, and $r[s_1, s_2]$ is the transition rate from state s_1 to s_2. Other steady state queries are computed similarly. In terms of the functions $f_n(\cdot)$, let $f_i(\cdot) = 0$ for $0 \leq i < N$ and

$$f_N(X^N(s)) = \frac{e^{\Lambda t} N!}{(\Lambda t)^N} (\vec{r} \cdot X^N(s))$$

where $\|X^N(s) - X^{N-1}(s)\| < \epsilon$.

Pruning of States

Each element of $X^n(a, s, \vec{k})$ contains the probability of the augmented Markov chain being in state $(a, s, \vec{k})$ after n transitions. Our experience has shown that these probabilities $X^n(a, s, \vec{k})$ typically have a range of many orders of magnitude for even moderately sized n. There are two reasons for this behavior. First, there is typically a range of values along each row of the matrix describing the Markov chain. As n gets large, there is some small probability that a sample path of the Markov chain may traverse all the unlikely states. Consider a Markov chain where each row has only two non-zero elements, 1/4 and 3/4. A sequence that takes the lower probability (1/4) transition at each opportunity will be almost 4 orders of magnitude less probable than the the sequence that takes the higher probability transitions after only 8 steps. As the transition probabilities typically exhibit a much larger range of probabilities than from 1/4 to 3/4, this problem is greatly increased. The second reason that the range of $X^n(a, s, \vec{k})$ tends to be great is that for many applications the low probability sequences are often much less likely to be aggregated than other sequences. This can be see by looking at a simple reliability example with repair. The system being modeled can be in one of two states, operational represented by a reward of 1 or non-operational represented by a reward of zero. Looking at this system for 10 steps will generate up to $2^{10} = 1024$ different reward sequences. These reward sequences can be aggregated into 11 different buckets, each representing one of the possible number of operational states (from 0 to 10) in the reward sequence. The number of sequences aggregated into each bucket, 10 choose i ($_{10}C_i$) in the ith bucket, is shown in the following table.

bucket	0	1	2	3	4	5	6	7	8	9	10
number	1	10	45	120	210	252	210	120	45	10	1

It is clear that the number of sequences aggregated into each bucket varies considerably. Typically the low probability states are the ones with the small numbers of operational states which also are aggregated with few other states.

If those states of $X^n(a, s, \vec{k})$ with extremely small probability are pruned from further calculation, a savings in both computer memory and processing time would result with little change in the final results. Further, since the probabilities of the sequences being pruned are known, the total discarded probability would be an upper bound on the error introduced due to pruning.

In addition to the error introduced by pruning, there is also a truncation error introduced by truncating the infinite summation in randomization at some finite N. Thus if $P(> N)$ is defined to be the probability that more than N events of a Poisson process with rate Λ

occur in time T

$$\begin{aligned} P(> N) &= \sum_{k=N+1}^{\infty} e^{-\Lambda T} \frac{(\Lambda T)^k}{k!} \\ &= 1.0 - \sum_{k=0}^{N} e^{-\Lambda T} \frac{(\Lambda T)^k}{k!} \end{aligned}$$

then given some ϵ and given some bound $\bar{f}$ on the functions $f_n(\cdot)$, $N(\epsilon)$ can be chosen so that the truncation error due to randomization is less than ϵ.

$$N(\epsilon) = \min_N \left(P(> N) < \epsilon / \bar{f} \right)$$

In order to keep the sum of the pruning error and the truncation error less than ϵ, a portion of ϵ, say ϵ_p, can be allotted for the pruning error and the remainder, $\epsilon_t = \epsilon - \epsilon_p$ for the truncation error.

In order to bound the error due to pruning, we divide a computation up into *basic computation steps.* Each basic computation step takes one combined Markov chain state triple $(A, S, \vec{K})$ and finds a new state that can be derived from it. Thus Figure 10 (discussed earlier) shows two basic computation steps. Given a time T, a limited number of basic computation steps can be executed (depending on the speed of the computer), say $\mathcal{Z}(T)$. Thus in time T, any sequence with probability less than $\epsilon_p / \mathcal{Z}(T)$ cannot cause the total accumulated error from pruning (in time T) to be greater than ϵ_p. Unfortunately, a computation may not be finished with respect to the truncation error within time T. Since the exact amount of probability pruned is known, and is less than the total pruning error allotment, the computation can be continued at the users option with the remaining pruning error allotment.

One final problem is that meeting the pruning error allotment might mean that additional terms of the randomization summation have to be computed, possibly as many as $N(\epsilon_t) - N(\epsilon)$ extra terms. The number of extra terms, if any, is likely to be small for two reasons. First, from the formula above it can be seen $N(\epsilon)$ increases slowly as ϵ gets smaller in the typical area of interest; specifically when ϵ is near zero. Secondly, at any point in the calculation it is known exactly how much probability has been pruned, and therefore the calculations can often be curtailed before $N(\epsilon_t)$. For example, if $\epsilon_p/2$ probability has been discarded due to pruning, the summation can be truncated after $N(\epsilon_t + \epsilon_p/2)$ iterations.

5 Conclusion

We are in the process of implementing this system. We currently have the language translator which generates the Mealy machine and the following solvers: distribution of cumulative

time, expected total reward, expected cumulative time, transient point reward, and steady state analysis. With the exception of the distribution of total reward, the remaining solvers will not be difficult to add on to the existing base, and we are currently doing so.

We have shown how Markov chain queries can be represented in a very high level language. We have shown how this language can be converted into a form that can be efficiently executed in a procedural language such as C. Finally, a number of examples were shown to demonstrate how the language is used.

A BNF for the Pattern Reward Function

The BNF for the pattern reward function language is described below. Left and right angle brackets (< and >) are used to denote nonterminals while italics denote terminals. For simplicity <identifier> and <number> are not defined but take on their usual meaning.

<reward_function> ::= <reward_clauses>

<reward_clauses> ::= <reward_clause> |
<reward_clause> <reward_clauses>

<reward_clause> ::= <identifier> (<pattern>) *where* <predicate_list> . |
<reward_clause> ::= <identifier> (<simple_pattern>) *yields* <number> *where* <predicate_list> .

<pattern> ::= <identifier> |
<identifier>:<number> |
(<pattern>) |
<pattern> * |
<pattern> + <pattern> |
<pattern> , <pattern>

<simple_pattern> ::= <identifier> |
(<simple_pattern>) |
<simple_pattern> * |
<simple_pattern> + <simple_pattern> |
<simple_pattern> , <simple_pattern>

<predicate_list> ::= <predicate> |
<predicate> , <predicate_list>

<predicate> ::= <identifier> (<identifier_list>)

<identifier_list> ::= <identifier> |
<identifier> , <identifier_list>

References

[AHO74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[BERS90] S. Berson, E. De Sousa e Silva, and R. Muntz, "A Methodology for the Specification and Generation of Markov Models," *Proceedings of the First International Conference on the Numerical Solution of Markov Chains*, January 1990.

[CLOC87] W.F. Clocksin, and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, 1987.

[DAVI81] H.A. David, *Order Statistics*, Wiley, 1981.

[DESO86] E. de Souza e Silva and H. R. Gail, "Calculating Cumulative Operational Time Distributions of Repairable Computer Systems," *IEEE Transactions on Computers*, April 1986.

[DESO89] E. de Souza e Silva and H.R. Gail, "Calculating Availability and Performance Measures of Repairable Computer Systems Using Randomization," *Journal of the Association for Computing Machinery*, January 1989.

[GOYA86] A. Goyal, W.C. Carter, E. de Souza e Silva, S.S. Lavenberg, and K.S. Trivedi, "The System Availability Estimator," *Proceedings of FTCS-16*, July 1986.

[GROS84] D. Gross and D.R. Miller, "The Randomization Technique as a Modeling Tool and Solution Procedure for Transient Markov Processes," *Operations Research*, March-April 1984.

[HARR65] M.A. Harrison, *Introduction to Switching and Automata Theory*, McGraw-Hill, 1965.

[HOPC79] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[MAKA82] S.V. Makam, and A. Avizienis "ARIES 81: A Reliability and Life-Cycle Evaluation Tool for Fault Tolerant Systems," *Proceedings of FTCS-12* June 1982.

[SAHN87] R. A. Sahner and K. S. Trivedi, "Reliability Modeling Using SHARPE," *IEEE Transactions on Reliability*, June 1987.

[SAND91] W. H. Sanders and J. F. Meyer, "A Unified Approach for Specifying Measures of Performance, Dependability and Performability," *Dependable Computing and Fault-Tolerant Systems*, Springer-Verlag, 1991.