AN OBJECT-ORIENTED METHODOLOGY FOR THE SPECIFICATION OF MARKOV MODELS

Steven Berson
Edmundo Silva
Richard Muntz

July 1987
CSD-870030

AN OBJECT ORIENTED METHODOLOGY FOR THE SPECIFICATION OF MARKOV MODELS

S. Berson †

E. de Souza e Silva ††

R.R. Muntz †

† UCLA Computer Science Department

Los Angeles, CA 90024

†† PUC/RJ Electrical Engineering Department

Rio de Janeiro, Brazil

April 1987

AN OBJECT ORIENTED METHODOLOGY FOR THE SPECIFICATION OF MARKOV MODELS

ABSTRACT

Modelers wish to specify their models in a symbolic, high level language while analytic techniques require a low level, numerical representation. The translation between these description levels is a major problem. We describe a simple, but surprisingly powerful approach to specifying system level models based on an object oriented paradigm. This basic approach will be shown to have significant advantages in that it provides the basis for modular, extensible modeling tools. With this methodology, modeling tools can be quickly and easily tailored to particular application domains. An implementation in Prolog, of a system based on this methodology is described and some example applications are given.

1. Introduction.

The complexity of the new generation of highly concurrent systems that are now being developed has made the use of sophisticated modeling tools for specification and analysis a high priority enterprise. The variety of architectures and different problems to be analyzed demonstrate the need for general tools, i.e., tools that allow specification of general classes of models. There are many examples of such tools in the literature [Saue81, Saue84, Berr82, Goya86, Triv84, Maka82, Cost81, Carr86]. The usefulness of such tools can be measured in terms of two factors: the sophistication of the underlying analytic and/or simulation techniques used and the simplicity and power of the user interface. Development of analytic techniques and their implementation in analysis packages is not enough; to be truly useful the analysis must be made easily accessible to the modeler. Unfortunately, existing systems are limited in usefulness due to the user interfaces that are provided. This paper addresses exactly this problem.

The problem is one of mapping between two representations of a model: *the modelers representation* and the *analytic representation*. The analytic representation is the detailed, low level representation required as input to the analysis modules. The modelers representation is the model specification that the user supplies. The modelers representation is typically in symbolic form and should be in terms of constructs that are natural to the application. Clearly the form of the model specification language determines the ease of defining models thereby influencing the overall cost of the modeling effort. A high level language tailored to a particular application domain provides the ability to define models that are easy to understand, less time consuming for the modeler and less error prone.

Analytic techniques are often general purpose and applicable to a wide range of problem domains. It is desirable to make the analysis tools easily accessible for these varied applications. A fundamental problem is that the most suitable model specification language will vary from one application to another. A "one language suits all" approach is therefore not appropriate.

This is however the approach taken by current tools. The result is one of two extremes: (1) a language that is specific to a narrow application area or (2) a language that achieves generality by providing only primitive constructs (e.g. stochastic Petri nets). Both tend to hinder access to the analytic tools available. The first provides a convenient user interface for models that fit into the anticipated mold but for no others. The second places too much of the burden on the modeler. The paucity of modeling constructs often forces the modeler to build descriptions that are contrived, complex and unrelated to any "natural" representation. What is required is a methodology that permits the tailoring of the model specification language to the application domain. This paper presents such a methodology.

In this paper we concentrate on modeling applications for which the analytic representation is a Markov process state transition rate matrix. This covers a broad class of applications as Markov processes are the most general representation typically provided for representing and solving performance and reliability models. Due to the decrease in the cost of memory, the increase in computation power, and recent advances in solution techniques, Markov processes with tens of thousands of states can now be solved; somewhat easing the large model problem, which is a major limitation.

The conceptual basis of our approach is an object oriented paradigm for model description. In this approach all system models are defined in terms of instances of objects and interactions between objects. The characteristics of specific applications are reflected in the object types that are used in the model specification. By providing libraries of object definitions the system can be easily extended to particular application domains. System models that combine object types drawn from a library with user defined object types are accommodated, which permits specialized extensions. As will be demonstrated by the examples that appear later, this has proven to be a powerful modeling paradigm that has easily accommodated a wide variety of applications including reliability models and queueing theoretic models. With this approach we have been able to produce tailored interfaces in a matter of hours for reliability modeling and queueing networks while other methods require weeks of effort. Experiences with defining ad hoc models indicate similar efficiencies. Being able to easily construct and analyze specialized models is of significant benefit in a research environment. The effort required with existing systems has previously discouraged exploratory studies of specialized models.

This research has been influenced and has drawn from a variety of sources and several of the most related efforts are noted here. The METFAC system [Carr86] uses production rules to describe system behavior. This system does not utilize an object oriented approach and the production rules operate on the global system state. The object oriented approach has the advantage of modularity and leads to natural support for higher level interfaces in which a particular model is specified in terms of previously defined objects.

Lenders [Lend85] developed a method of describing distributed computations using Prolog. He showed how to model distributed computations with communicating finite state automata in Prolog and how to generate the reachable set of states. His purpose was to analyze algorithms for liveness and safety properties. We use the same general approach to generate the set of states reachable from an initial state, but extend the general method to account for a number of special features desirable for performance and reliability modeling.

In section 2 we describe the system modeling paradigm independent of any particular implementation. In section 3 we describe the organization of the system that has been constructed to implement the approach. Section 4 contains several example applications and section 5 presents our conclusions.

2. The Object Oriented Modeling Paradigm.

The system description paradigm that we have adopted is *object oriented.* In this view a system is composed of a set of interacting components called **objects**. Each object is an entity that has an internal state and can evolve over time. In addition, objects can generate actions called **events** at some rate. The state of an object will determine the types of events it can generate and the rates at which they occur. The generation of an event can also be conditioned on the state of other objects. This is expressed as a boolean function (predicate) on the global system state. An event may simply cause the object generating it to change state with no effect on other objects but in general, an event will cause other objects to react in some manner. This is modeled by allowing objects to generate **messages** to be *broadcast* to other objects notifying them of an event. The specification of an object includes a definition of how it reacts (i.e. changes state and sends messages) to received messages.

This object oriented paradigm has the following advantages:

a. it is natural to think of a system as a set of interacting components.

b. the paradigm introduces modularity into the system description since object behavior is conceptually divided into internal behavior of the object and the interaction with other objects (via messages) and via the preconditions on events.

Example

As an **example,** a system for reliability modeling could be described in terms of two types of objects: system component and a repair service. The "internal" state of a component may be "operational" or "down". An "operational" component can generate a "fail" event with some specified rate. This event may affect other objects, e.g. other components may be caused to fail, and the repair service object may change state to record the newly failed components queued for repair. The effects of a component failure on these other objects is represented by a message sent to these objects and their reactions to the message.

The object oriented paradigm is intended to provide a generic model specification "schema." The idea then is to provide a tool that can translate any model instance expressed in the schema framework to an analytic representation. A particular implementation of the basic idea is distinguished by the language used to specify objects, events, etc.

Simultaneous Events

The semantics of a model expressed in the above paradigm are fairly obvious except for the occurrence of messages that "propagate". Notice that in the definition of the reaction of an object to a message the object may in turn generate further messages. The intended meaning is that messages are delivered (and reacted to) instantaneously. In essence this can result in a cascading effect in which multiple changes in state occur at the same instant. The usefulness of this notion is illustrated in the example given above in which a component that fails can cause other components to fail. This is the weakest point in the paradigm with respect to conceptual clarity. Similar problems arise in other modeling schemes that allow "zero time" actions, e.g. Generalized Stochastic Petri Nets [Mars84]. In the discussion following the description of our implementation we will return briefly to this issue.

Object Types

An obvious extension to the basic paradigm is to add the notion of object types. This is more in the nature of an implementation issue and simply allows for economy in specifying a model. This will be incorporated into the implementation described in later sections.

We have clearly borrowed notions from object oriented programming but it must be emphasized that we are describing a conceptual framework for describing system behavior and not an implementation language. A system description in the framework described above is declarative in nature and not a program to be executed. The description must be *interpreted* to generate the corresponding Markov process state transition representation. Interpretation of the description is essentially a search of the reachable system states starting from a specified initial state.

Our experience indicates that the object oriented framework for description of a system is very general and can be applied to any modeling application for which it is desired to generate the Markov state description. In fact, other target representations are possible and are under investigation.

3. Implementation Organization.

In the following sections we describe an implementation of the concepts in the previous section. Prolog was chosen as the implementation language for both object definitions and the "interpreter" of these definitions. Prolog was chosen for several reasons. The ability to use the same language for both purposes leads to a simpler and more concise implementation. Prolog allows simple, declarative descriptions of objects and permits the specification of complex rules of behavior in a simple, easy to understand manner. For the reader unfamiliar with Prolog, a summary of relevant features is given in Appendix A.

In order to facilitate use of the tool and aid in tailoring it to particular applications we have distinguished four different "layers" in its organization, as shown in figure 1.

a. The core of the tool deals with any object oriented description and generates the Markov state description. This part is independent of the particular application.

b. The next layer is the definition of object types. Object types will in general be application dependent and to formulate the descriptions will require some knowledge of Prolog. It is expected that object definitions will be parameterized so that instances of the objects can be declared and parameters specified for each instance. Libraries of object type definitions would be created so that a user could define an instance of an object type by simply referring to the library definition.

c. The object definitions provided in layer "b" allow an "end user" to define a model by declaring instances of the objects and supplying the required parameters. These declarations are in the form of simple Prolog clauses that are referred to in the procedures defining the object type. At this level one defines an instance of a model by declaring the object instances and for each the object, the parameters.

d. Finally, if one wishes to provide a sophisticated user interface (e.g. more English-like or graphical), this can be built on top of layer "c" by providing the translation from the end user interface to the Prolog clauses. We have not concentrated on this layer thus far, although we have written a translator for the SAVE [Goya86] user interface to Prolog.

3.1 Core Interface.

The core of the system provides a generic interface which takes descriptions of objects, events and an initial state and generates the Markov state description. The "interface" to the core is the basis on which everything else is built. The core assumes the following "schema" or format.

(1) each object has a name which is arbitrary but distinct.

(2) each object has a set of possible states. The core makes no interpretation of the states of an object.

(3) the "global state" of the system is a list containing one entry for each object. The entry for an object is a 2-tuple containing:

a) the name of the object,

b) its current state.

(4) for each object, a definition of the events that the object can generate for each local state of the object and the rate at which that event occurs.

(5) for each object, a definition of the possible effects of an event generated by some other object and for each possible effect, the probability that that effect occurs.

(6) initial state: a statement of the initial global state of the system.

The core operates on this description by searching for the states reachable from the given initial state. A standard recursive search procedure is used that, for each reachable state S, determines the states reachable from S by some event given that the system is in state S. Any search procedure (e.g. breadth first or depth first) is sufficient.

We now describe in more detail the interface to the core as it has been implemented. Interfacing with the system at this level of detail is only required of a "library designer." For general classes of models it is intended that this would be done once and "end users" would simply refer to the library definitions. After deciding the types of objects in a system, the library designer has to list the events which can be generated by each object. Furthermore, the conditions under which these events can occur must be specified. Typically, this is a function of the object's local state, but may depend on the current system state as well. Next, the "reaction" of an object for each type of event has to be defined, including the specification of messages that may be sent to other objects. Finally, the "reaction" of objects to receiving a message are specified and, if different reactions are possible, the probability of each occurrence. In summary, the core expects the following predicates to be specified for each object:

a. event(**Object, Event**).

b. valid_event(Object, Event, Local_state, Global_state).

c. react(Object, Event, Local_state, Global_state, **Next_Global_state, Msgs, Rate**).

d. message(Message, Old_state, State, **New_state, Probability, Msgs**).

Predicate "event" indicates the possible events which can be generated by an object. Predicate "valid_event" succeeds if the object can generate *Event* in the current state of the model. The "react" predicate describes the reaction of the object to an event that it generates. A reaction consists of changing from one global state to the next and sending out messages, if any. The "message" predicate describes how an object reacts to messages that come from other objects and with what probability.

3.2 Example of Object Type Definition.

In this section we illustrate the definition of a set of object types for a simple repair model in which components can fail independently and are repaired in a "first-come-first-serve" (FCFS) order. In availability models, components of the same type are generally clustered together. If there are some number of CPUs with identical failure/repair behavior, it is easier (and generates fewer states) if they are treated as a CPU cluster with the state being the number that are operational. We choose to represent the system with two types of objects: a *components* type object which models components clusters and a *repair facility* type object. There are two types of events: *failure* events which are generated by objects of type *component*, and *repair* events which are generated by a *repair facility* object. The state of an object of type *component* is the number of operational (Up) units. The state of the *repair facility* object is a FCFS queue. The Prolog clauses for specifying the object types are given below.

```
event(Object_name, failure) :-
        component(Object_name, _).
event(repair_facility, repair).

valid_event(Component, failure, Local_state, Global_state) :-
        Local_state > 0.
```

```
valid_event(Repair_facility, repair, [In_Repair| In_Queue], Global_state).

react(Component, failure, Local_state, Global_state, Next_global_state, Rate, Msgs) :-
        Msgs = [[repair_facility, failure, Component]],
        New_local is Local_state - 1,
        update_state(Component, New_local, Global_state, Next_global_state),
        failure_rate(Component, Base_rate),
        Rate = Local_state*Base_rate.

react(Repair_facility, repair, [Served|In_Queue], State, Next_state, Rate, Msgs) :-
        Msgs = [[Served, repaired]],
        update_state(Repair_facility, In_Queue, State, Next_state),
        repair_rate(Served, Rate).

message([repair_facility, failure, Component], Old_state, State, New_state, 1, []) :-
        member([repair_facility, Queue], Old_state),
        append(Queue, [Component], New_queue),
        update_state(repair_facility, New_queue, State, New_state).

message([Component, repaired], Old_state, State, New_state, 1, []) :-
        member([Component, Up], Old_state),
        New_up is Up + 1,
        update_state(Component, New_up, State, New_state).
```

The first *event* clause indicates that an event of type *failure* is generated by any component, i.e. any component can fail. The second *event* clause indicates that the object *repair_facility* can generate an event of type *repair*.

The next two clauses validate the events. The first *valid_event* clause indicates that a component type may generate a failure event if the number of operational components of that type is greater than zero (Local_state > 0). The second clause states that a repair can occur if the repair facility is non-empty, i.e., there are components to be repaired.

The two *react* clauses describe how each object reacts to events it generates (i.e. changes state). The first *react* clause describes the reaction of an object of type component to a failure. First, the message, [repair_facility,failure,Component], is created to be sent to the repair facility requesting that it repair the failed component. Second the number of operational components is decremented. This will be the new local state of "Component." Then the global state is updated to reflect the failure. The *failure_rate* clause is a parameter routine provided by the user to specify the failure rate of each component. Finally, the rate is calculated by multiplying the number of operational components by the base failure rate for that component. The second *react* clause describes the reaction of the repair facility when a repair event occurs. First, the component which is at the head of the repair queue (component being repaired) is extracted from the local state. Then a message to the repaired object is generated. The new local state will be the remainder of the queue. Finally, the new local state replaces the old local state in the global state and the rate of occurrence of this event is calculated. The *repair_rate* predicate is similar to *failure_rate* and provides the generic clause the rate of repair for each component.

Message reactions also have to be described. The first *message* clause describes the reaction of the repair facility upon receiving a message that some particular "Component" has failed. No messages are generated in response to this message. Then the queue of the repair facility is extracted from the old global state (*member* is a system defined lookup predicate). Next the new local state is formed by adding the component which failed to the tail of the queue. Finally the old local state is replaced by the new local state in the global state. The second *message* clause describes the reaction of an object of type "Component" upon receiving a message indicating that one of these type of components was repaired. The meaning should be clear from the clause.

3.3 Using Predefined Object Types.

The previous section presented a complete description of object types for a simple reliability modeling system. To specify a particular model instance, the user needs to indicate the components of the system, the failure and repair rates, and finally the initial state. Note that the effort involved in defining the object definition library is done once. To define a specific model instance is quite simple. For example, assume that we want to specify the model for a system with one CPUs and two memories. The user would load the library of object definitions described above and add the following few statements.

```
component(cpu,1).
component(memory,2).
failure_rate(cpu,cpuFrate).
failure_rate(memory,memoryFrate).
repair_rate(cpu,cpuRrate).
repair_rate(memory,memoryRrate).
initial([[cpu,1],[memory,2],[repair_facility,[]]]).
```

The output from this example is included below. First the states and state numbers are listed, then the state transition rates.

```
0       [[cpu,1],[memory,2],[repair_facility,[]]]
1       [[cpu,0],[memory,2],[repair_facility,[cpu]]]
2       [[cpu,1],[memory,1],[repair_facility,[memory]]]
3       [[cpu,0],[memory,1],[repair_facility,[memory,cpu]]]
4       [[cpu,1],[memory,0],[repair_facility,[memory,memory]]]
5       [[cpu,0],[memory,1],[repair_facility,[cpu,memory]]]
6       [[cpu,0],[memory,0],[repair_facility,[cpu,memory,memory]]]
7       [[cpu,0],[memory,0],[repair_facility,[memory,memory,cpu]]]
8       [[cpu,0],[memory,0],[repair_facility,[memory,cpu,memory]]]
```

```
0->1    Rate:cpuFrate
0->2    Rate:2*memoryFrate
2->3    Rate:cpuFrate
2->4    Rate:memoryFrate
2->0    Rate:memoryRrate
1->5    Rate:2*memoryFrate
1->0    Rate:cpuRrate
5->6    Rate:memoryFrate
5->2    Rate:cpuRrate
4->7    Rate:cpuFrate
4->2    Rate:memoryRrate
3->8    Rate:memoryFrate
3->1    Rate:memoryRrate
8->5    Rate:memoryRrate
7->3    Rate:memoryRrate
6->4    Rate:cpuRrate
```

Figure 2 illustrates the state transition diagram.

3.4 Discussion.

This section contains a discussion of several topics that were omitted from the general description.

3.4.1 Trap States

There are cases for which it is desirable to define trap states for the model. A trap state is a state with no transitions out of it. For example, in availability models one would want to define the conditions under which the system is considered to have failed. It is also convenient to be able to truncate the state space by not allowing more than some number of failures. These cases are handled by allowing the definition of trap states. The trap predicate allows the user to define what system states correspond to trap conditions. If we want to trap on states with two failures in the example in Section 2.3 we would add:

```
trap(State, trap_state_name) :- member([repair_facility, [_, _]], State).
```

While searching the state space, a transition to a state S for which trap(S) is true, is automatically changed to a transition to a state *trap_state_name*. A more complex example is given in the section 4.

3.4.2 Querying the Analysis Results

The numerical analysis routines return the equilibrium state probabilities for all states. The system provides a way for intelligently querying these results. Predicates are supplied that associate a reward with each state. If no reward is supplied, then zero is assumed. The product of the rewards and the associated state probabilities are summed and the result returned. In the example below the *unbalanced* predicate associates a reward of one with all states that have a customer waiting in one queue while the other other server is idle.

```
unbalanced(State,Reward) :-
```

```
        member([cpu1,Number],State),
        member([cpu2,0],State),
        Number > 1,
        Reward = 1.
unbalanced(State,Reward) :-
        member([cpu1,0],State),
        member([cpu2,Number],State),
        Number > 1,
        Reward = 1.
```

The actual query would be:

```
query(unbalanced,Result).
```

Similar to objects, libraries of queries can be built that a user could invoke.

3.4.3 Modularity of Object Definitions

One would like to have interactions among objects be completely described in terms of the messages that are sent between objects. We have found it more convenient to allow objects to examine the global state of the model in the procedures that define their behavior. To clarify the problem, suppose that we are defining objects for a reliability model. Suppose also, that an object is "dormant" if some other set of objects have failed. In the dormant state the failure rate of an object can be different than when it is operational. The failure rate of this object depends on the state of these other objects. There are two ways that this could be accounted for in the object definitions:

1. When an object fails or is repaired, then a message is sent to all other objects. Each object can interpret these messages and keep track of whether it is dormant or not.

2. Each object does not explicitly keep track of whether it is dormant or not. This can be determined "dynamically" in the procedures defining its behavior by examining the state of the other objects. This is done by sending a message to another object to request state information.

There are obviously tradeoffs between the two approaches. The first is "cleaner" conceptually but seems more clumsy to implement. We have favored the second approach in our system.

3.4.4 Simultaneous Actions

The manner in which objects and messages are modeled permits simultaneous actions by objects. For example a message may be sent "simultaneously" to several objects. In the implementation these messages are "delivered" in an unspecified order. The implementor must be aware of this and be sure that the order of delivery is not relevant, i.e. does not effect the state transitions.

As an example of the type of problem that can arise, consider the case of availability modeling. When a component fails it can send a message to another component that may be affected (as well as the repair service object). One action of the effected component may be to fail, and in this case it should send a message to the repair service object as notification. To describe the rules applied to the "delivery" of propagated messages we can consider a rooted tree representing the generation of messages. The nodes of the tree represent objects and the arcs represent messages sent. At the root is the object that generated the original event. In terms of this tree, messages are delivered level by level.

In most cases, the ordering of message delivery has no effect on the state transitions. Only when two simultaneous messages go to the same component is there a potential ambiguity. Even then, there is no problem with many disciplines. Objects with processor sharing queueing disciplines are not affected by simultaneous arrivals. Neither are objects with priority nor infinite server disciplines.

There is one other issue. When a message is received, does it have access to the global state of the system prior to the event that started the transition or does it have access to the changes that have been made to the global state by messages that have already been delivered?

In our system, when a message is delivered, a message comes with both the previous global state and the current global state. This allows the object to examine either global state.

4. Examples.

In this section we present several examples which further illustrate use of the system. First we show an example of a non-trivial trap rule. Second, we consider defining object types appropriate for a queueing network interface.

4.1 Data Availability Modeling Example.

A simple example should serve to illustrate the ease of representing a useful, non-trivial system model feature in Prolog. Suppose we have a distributed architecture model as illustrated in figure 3. The "connectivity" between components can be represented by a set of Prolog "facts".

```
connected(State, cpu1, bus) :- up(State, cpu1), up(State, bus).
connected(State, cpu2, bus) :- up(State, cpu2), up(State, bus).
connected(State, bus, controller1) :- up(State, bus), up(State, controller1).
connected(State, bus, controller2) :- up(State, bus), up(State, controller2).
connected(State, controller1, disk1) :- up(State, controller1), up(State, disk1).
connected(State, controller2, disk1) :- up(State, controller2), up(State, disk1).
connected(State, controller2, disk2) :- up(State, controller2), up(State, disk2).
```

These rules state that a direct data path exists between the named components if both components are operational ("up") in the current state.

A rule which defines the existence of a data path between two components can be given by the following rules.

```
path(State,X,Y) :- connected(State,X,Y).
path(State,X,Y) :- connected(State,X,Z),path(State,Z,Y).
```

The first rule states that there is a path from "X" to "Y" if "X" and "Y" are directly connected.

The second rule states that there is a path from "X" to "Y" if "X" is directly connected to some component "Z" and there is a path from "Z" to "Y".

Now suppose that critical data is replicated as described by the following facts:

```
copy(d1, disk1).      % copy of data item d1 on disk1
copy(d1, disk2).      % copy of data item d1 on disk2
copy(d2, disk2).      % copy of data item d2 on disk2
```

A rule can be used to define the availability of a data path to at least one copy of each data item:

```
data_available(State) :- data_available(State,d1), data_available(State,d2).

data_available(State,d1) :- copy(d1,Disk),cpu(CPU),path(State,CPU,Disk).
data_available(State,d2) :- copy(d2,Disk),cpu(CPU),path(State,CPU,Disk).

trap(State, trap_state) :- not( data_available(State) ).
```

The first rule states that all data is accessible if both data items "d1" and "d2" are accessible. The next two rules state that the conditions for availability of each data item. For example, "d1" is available if there is a disk containing a copy of "d1" and a processor with a path from the processor to the disk. The last rule says that the system is no longer operational if there is some data that is not available.

The above description is one example of how relationships and rules of behavior can be represented relatively simply in Prolog.

4.2 Queueing Network Example

In this example we show the ease of adding new object types in conjunction with object types from a library. We emphasize the ease with which new objects can be created and merged in with predefined objects. This system is a simple load balancing system as shown in figure 4.

There is a set of terminals, a scheduler, and two CPUs. The terminals use the infinite server discipline, and the CPUs use the processor sharing discipline. Both these disciplines are part of the queueing systems standard library of types. The shortest queue scheduler, however, is not part of the basic library and must be specified.

```
initial([                           % Initial global state
        [terminals,[3]],
        [scheduler,[0,0]],
        [cpu1,[0]],
        [cpu2,[0]]
]).

type(terminals, inf).               % object instance declarations
type(cpu1, ps).
type(cpu2, ps).

route(terminals, scheduler, 1).          % object instance declarations
route(cpu1, terminals, 1).
route(cpu2, terminals, 1).

departure_rate(terminals, tr). % object instance declarations
departure_rate(cpu1, cr).
departure_rate(cpu2, cr).
%
%             Start of new object type definition
%
event(scheduler, update).

update_rate(cpu, ur).

valid_event(scheduler, update, _, _).

react(scheduler, update, Local_State, State, New_State, Rate, []) :-
        update_rate(scheduler, Rate),
        member([cpu1,[Num1]], State),
        member([cpu2,[Num2]], State),
        subst(scheduler, [Num1,Num2], State, New_State).

message([scheduler,arrival], _, State, State, 1/2, [[cpu1,arrival]]) :-
        member([scheduler,[X,X]], State).

message([scheduler,arrival], _, State, State, 1/2, [[cpu2,arrival]]) :-
        member([scheduler,[X,X]], State).

message([scheduler,arrival], _, State, State, 1, [[cpu1,arrival]]) :-
        member([scheduler,[X,Y]], State),
        X < Y.
```

```
message([scheduler,arrival], _, State, State, 1, [[cpu2,arrival]]) :-
        member([scheduler,[X,Y]], State),
        Y < X.
```

The initial state of the system has three customers at the terminals, and no other customers in the system. The state of the scheduler has what the scheduler thinks are the queue lengths of the CPUs. Cpu queue length updates are sent to the scheduler at the rate specified in the *update_rate* predicate. The terminals are defined as an infinite server queue (inf). Both CPUs are defined as processor sharing queues (ps). The routing from these nodes is specified. The CPU and terminal departure rates are specified. This completes the standard part of the model. The scheduler must now be defined.

The *update_rate* predicate sets the rate of CPU queue length updates. An update event is always valid. The react predicate gets the update rate. The member predicates lookup the number of customers in each of the CPUs. These queue lengths are then put in as the new state as seen by the scheduler.

There are four cases when the scheduler receives an arrival. In the first two cases, when the scheduler thinks that the queue lengths are equal, an arrival message is sent with probability 1/2 to either cpu1 or cpu2. In the third *message* predicate, the scheduler thinks the length of the cpu1 queue is less than the cpu2 queue, so an arrival message is sent to cpu1. The fourth clause is similar to the third. This is the entire specification for the system.

5. Conclusion.

The approach to defining system models and generating the Markov process description that has been described is very simple and yet powerful. It took only a few hours to define the "objects" for availability modeling of the same order of sophistication as the SAVE system. The same level of effort was required to define a set of objects for queueing network models. In addition we have had occasion to use the system to define various ad hoc models (e.g. a priority queuing system in which the low priority customers received service after waiting for some number of high priority customers). The ease with which these models were developed and solved surpasses any other system we know of.

The extensibility of the system is easily seen. As in the example, a model can easily be defined that incorporates object definitions from a library of predefined objects with new objects that the modeler wishes to define.

The emphasis in this system is on flexibility and power in defining models. A price paid is in execution efficiency. We have not yet carried out detailed performance studies but have found the performance to be acceptable. For example, to generate a one thousand state model on a VAX 780 took twenty minutes of cpu time. Improvements in performance are under study. However, we believe that the advantages will make the approach attractive for at least two purposes largely independent of the efficiency issue:

a. as a methodology for prototyping a new modeling interface, and

b. in defining ad hoc models for studies in which the main objective would be to quickly build the model rather than optimize the performance of the tool.

APPENDIX A

Prolog

Prolog is based on goals (or predicates) which evaluate to either true or false. Each goal is either a fact or is defined in terms of other goals. The format of a clause is

```
clause_head(arguments) :- goal1(args1), goal2(args2), ...goalN(argsN).
```

The ':-' is the Prolog symbol for 'if'. The above clause can be read "Clause_head(arguments) is true if all the goals, goal1(args1), goal2(args2), ... goalN(argsN), are true." There may be more than one clause of the same goal. If one clause fails, other clauses with the same clause head are tried in order to satisfy that goal.

Data in Prolog is untyped. Strings beginning with a capital letter or '_' are variables. Strings beginning with a small letter are constants. The '_' alone is a don't care condition. It will match with anything. A list can be represented in one of two ways. The first is by specifying all its elements in square brackets. The list with the elements 4,5,6 is [4,5,6]. The second way is using head and tail notation. The vertical bar separates the head and tail. The previous list would be written [4|[5,6]] in head and tail notation. The head and tail notation is similar to a first-come first-served queue. The head is the first element of the list. The tail is the remaining list. The null list, the list with no members, is written [].

When a goal is called, Prolog searches for a matching clause head with matching arguments. A constant matches only the same constant. A variable will match with a variable or a constant. A match between a constant and a variable will cause the variable to be bound to the constant. For example, if "likes(john, books)" is a clause, then the Prolog goal "likes(john,Variable)" will succeed with *Variable* bound to *books*.

One of the features of Prolog is that the variables in the clause head are not specified as input or output parameters. To make the Prolog in this paper more readable, we have used the convention that variables which are bound as a result of the call (output variables) are printed in bold face. Variables which will be bound (specified) when the goal is called are in normal type.

References

[Berr82]	Berry, R., K. M. Chandy, J. Misra, and D. M. Neuse, *Paws 2.0: Performance Analyst's Workbench Modelling Methodology and User's Manual*, Austin, Texas: Information Research Associates, 1982.
[Carr86]	Carrasco, J.A. and J. Figueras, "METFAC: Design and Implementation of a Software Tool for Modeling and Evaluation of Complex Fault-Tolerant Computing Systems," *Proceedings of FTCS-16*, July 1986, pp. 424-429.
[Cost81]	Costes, A., J. E. Doucet, C. Landrault, and J. C. Laprie, "SURF: A Program for Dependability Evaluation of Complex Fault-Tolerant Computing Systems," *Proceedings of FTCS-11*, June 1981, pp. 72-78.
[Goya86]	Goyal, A., 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 pp. 84-89.
[Lend85]	Lenders, P. M., *Modeling Distributed Systems with Logic Programming Languages*: Colorado State University Department of Mechanical Engineering, 1985. PhD dissertation.
[Maka82]	Makam, S. V. and A. Avizienis, "ARIES 81: A Reliability and Life-Cycle Evaluation Tool for Fault Tolerant Systems," *Proceedings of FTCS-12*, June 1982, pp. 267-274.
[Mars84]	Marsan, M. A., G. Conte, and G. Balbo, "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems," *ACM Transactions on Computer Systems*, May 1984, pp. 93-122.
[Saue81]	Sauer, C. H., E. A. MacNair, and J. F. Kurose, "Computer Communication System Modelling with the Research Queueing Package Version 2," IBM T. J. Watson Research Center, Yorktown Heigths, Tech. Rep. RA-128, November 1981.
[Saue84]	Sauer, C. M., E. A. MacNair, and J. F. Kurose, "Queueing Network Simulations of Computer Communication," *IEEE Journal on Selected Areas in Communications*, Vol. SAC-2, No. 1, January 1984, pp. 203-220.
[Triv84]	Trivedi, K. S., J. B. Dugan, R. R. Geist, and M. K. Smotherman, "Hybrid Reliability Modeling of Fault-Tolerant Computer Systems," *Comput. Elec. Eng.*, Vol. 11, 1984, pp. 87-108.

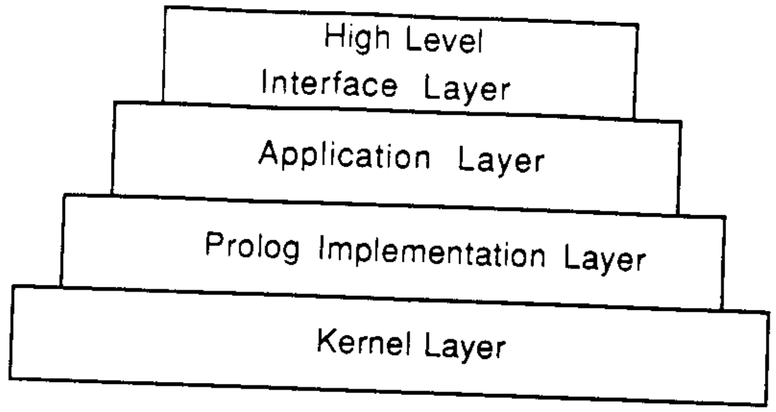

Figure 1 System Layers

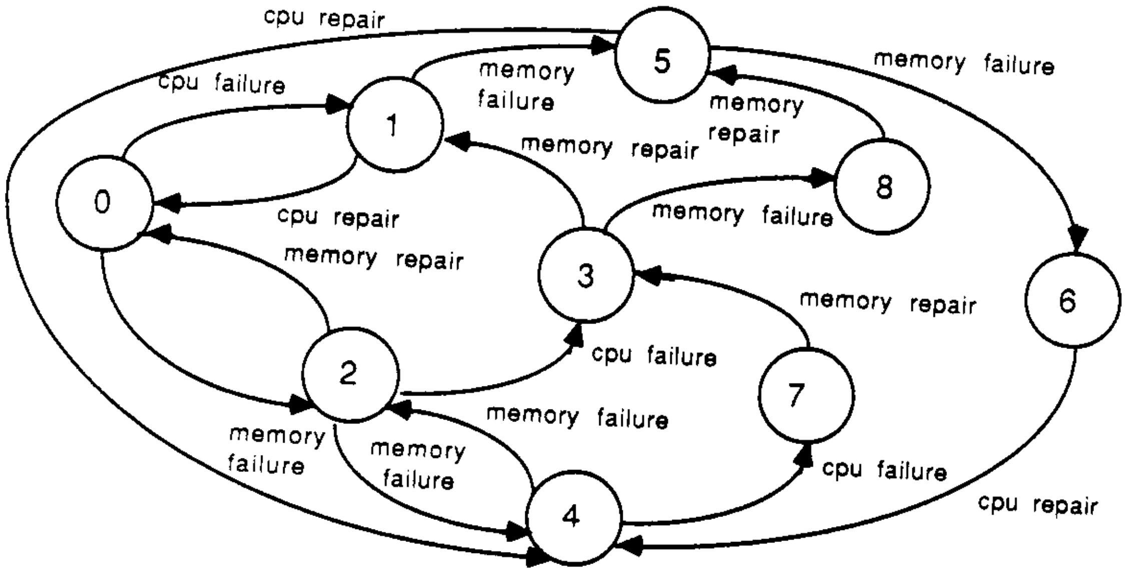

Figure 2 State Transition Rates

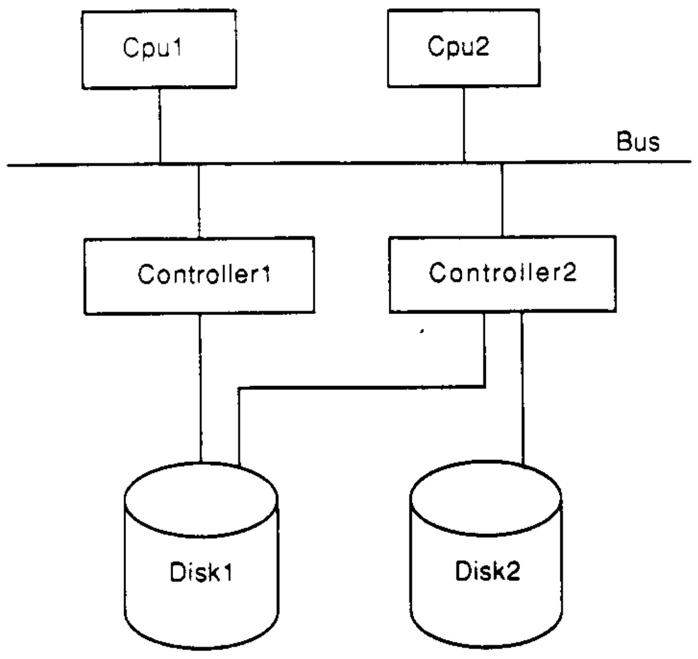

Figure 3 Network Connectivity

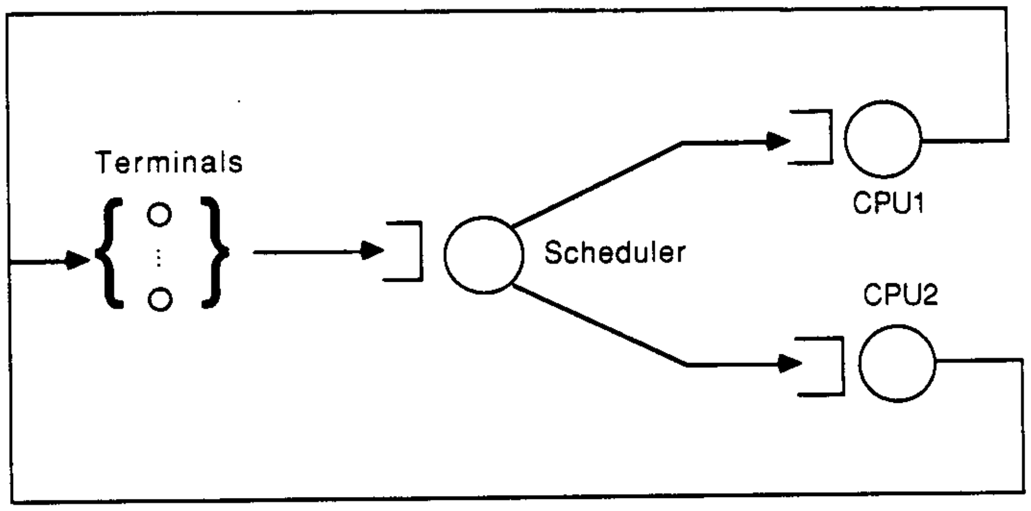

Figure 4 Queueing Network Example