AN OBJECT ORIENTED METHODOLOGY FOR THE SPECIFICATION OF MARKOV MODELS

S.BENSON *
E.DE SOUZA E SILVA
R.R.MUNTZ *

NCE-03/88

Abril/88

Universidade Federal do Rio de Janeiro
Núcleo de Computação Eletrônica
Caixa Postal 2324
20001 - Rio de Janeiro - RJ
BRASIL

* UCLA Computer Science Department

Esta pesquisa contou com o apoio financeiro da NSF-USA INT-8514377 e CNPq.
Este trabalho foi publicado como relatório técnico do Departamento de Ciência da Computação da UCLA CSD-870030, Junho 1987.

Resumo

É desejável a especificação de modelos de sistemas de computação em uma linguagem simbólica de alto nível. Entretanto, técnicas analíticas requerem uma representação numérica de baixo nível. A tradução entre estes níveis de descrição é um grave problema. Neste artigo, descrevemos um método simples, mas surpreendentemente poderoso para especificação de modelos a nível de sistema, baseado em um modelo orientado a objeto. Mostraremos que este método básico possue vantages significativas pois fornece a base para o desenvolvimento de ferramentas modulares que podem ser extendidas. Com esta metodologia, ferramentas de modelagem podem ser facilmente e rapidamente talhadas para um determinado domínio de aplicação. Uma implementação em Prolog de um sistema baseado nesta metodologia é descrita e alguns exemplos são incluídos. As vantagens de se usar Prolog como uma linguagem de implementação são também discutidas.

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 a 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. The advantages of Prolog as an implementation language are also discussed.

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. We are particularly concerned with the latter.

We concentrate in this paper on modeling applications for which the underlying 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. These advances have resulted in renewed interest in tools based on the numerical solution of the state transition rate matrix of the model. This is particularly true for applications in which the models require the representation of complex interactions among system entities and/or sophisticated control schemes as are found for example in reliability models and models of communication protocols and parallel architectures.

We can distinguish 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, i.e. some representation of the transition rate matrix. 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. For example, a reliability model might be expressed in terms of the number of each component type, their failure modes and rates, etc. while a multiprocessor system might be described in terms of the processors, buses and memory modules, their physical connection and the rates of processors accessing memory modules.

Clearly the form of the model specification language determines the ease of defining models thereby influencing the overall cost of the modeling effort; requiring the user to directly input the transition rate matrix for a model is certainly possible but would severely hinder the application of the tool. A high level model description 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. However this puts

the burden on the modeling tool to provide a *translation* from the modelers representation to the transition rate matrix. This translation can be likened to a programming language compiler that translates from a high level language to machine code.

A further complication is that the most suitable model specification language will vary from one application domain to another. A "one language suits all" approach is therefore not appropriate. Neither is a specialized language for one type of model. These are however the approaches taken by most of the current tools. The result is one of two extremes: (1) a language that is specific to a narrow application area (e.g. the fault tolerance analysis tool SAVE [GOYA86]) or (2) a language that achieves generality by providing only primitive constructs (e.g. stochastic Petri nets [MARS84]). Both approaches have significant problems. The first provides a convenient user interface for models that fit into the anticipated mode 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, with little effort, the tailoring of the model specification language to the application domain. This paper presents such a methodology.

The conceptual basis of the approach is an object oriented paradigm for model description. (For a discussion on "object-oriented" concepts, we refer to [STRO87].) 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 modeling tool can be easily extended to new 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 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 one is able to produce tailored interfaces in a matter of hours for reliability modeling and queueing networks. Defining ad hoc models exhibits similar efficiencies. Being able to easily construct and analyze specialized models can be of significant benefit.

This research has been influenced and has drawn from a variety of sources and several of the most related efforts are noted here.

As early as 1971 Irani and Wallace [IRAN71] investigated the specification of queueing network models using the notion of object types and object interactions (their terminology is different). In fact this is the earliest work that we are aware of that takes this approach to model specification and translation to the state transition matrix. They were particularly interested in the use of network diagrams to specify queueing network models for interactive graphics modeling tools. They recognized the need for a formal means of defin-

ing object types and the meaning of connections between objects in a network diagram. They developed the basic approach which permits the definition of object types and object interactions. In their system an object is defined in terms of set of matrices that describe the behavior of an object. In [WALL72] Wallace describes how the transition rate matrix for a network of interconnected objects can be automatically generated from the matrix descriptions of the individual components.

We have borrowed the basic concepts from Irani and Wallace in our system. There are however some major differences. One difference is in the method of describing individual object types. They suggested the use of matrices and a set of algebraic operations to construct the state transition matrix for the complete system model. We have used an approach in which the transition matrix for the complete model is incrementally constructed by searching for reachable states. This technique has several advantages. In many applications it is not feasible to generate the entire transition rate matrix due to the size of the state space. In such situations one may be content to truncate the state space e.g. eliminating states with more than some number of failed components is common in reliability models, particularly if error bounds can be obtained. Another advantage in the generative approach is that only reachable states are represented in the resulting transition rate matrix. Note that the set of states of the complete model is generally a proper subset of the cartesian product of the states of the individual objects. Finally, we note that although in theory the method proposed by Irani and Wallace could be extended to more general modeling applications, they limited their consideration to queueing network models and built in limitations to the type of interactions allowed between objects. The main disadvantage to our method is increased computational cost. We have traded this cost for generality. It is interesting to note that in programming terminology our technique uses "lazy evaluation" in the sense that object states and transition rates are only evaluated when needed. This allows the definition of objects with unbounded number of states, e.g. an M/M/1 queue. In a particular system model the number of states must of course be finite but this can be accommodated either by the model specification, e.g. a closed queueing network, or by truncating the state space as an approximation.

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. Our 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 generating the set of states reachable from an initial state but extended 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.

As noted by Irani and Wallace [IRAN71] a modeling system consists of a large amount of software including command interpreters, analysis packages and query and display facilities. This software represents a significant investment and one would like it to be applicable to a wide range of application areas; e.g. queueing network models, reliability models, etc. The objective is to develop an approach that allows a modeling tool to be tailored to different application domains while requiring that most of the software be reusable across application domains.

The approach we advocate is based on the observation that models are typically very modular in that the model consists of a set of component "objects" that are "connected" in some manner. This is exactly what comes to mind if one thinks of queueing network models, the objects being the queues. The approach then is to build a system that allows object types and object interactions to be defined in some standard manner which is interpreted by the (invariant) remainder of the software. Particular application domains are characterized by the object types that are applicable to that domain. Such a system is extensible since new object types can be declared and used in conjunction with previously defined object types.

We begin with a short informal description and then follow with a more formal description of the modeling paradigm. With the *object oriented* view that we have adopted, a system model is composed of a set of interacting components called *objects*. The interactions among objects are represented via a message based mechanism as will be described shortly.

Each object is an entity that has an internal state which can evolve over time. The state of an object can change due either to

1. an *event* which the object itself generates or
2. the receipt of a message from another object.

The state of an object will determine the types of events it can generate and the rates at which they occur. 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 *sent* to other objects notifying them of an event. The specification of an object therefore also includes a definition of how it reacts (i.e. changes state and sends messages) to received messages.

The state of the system is given by the set of states of the component objects and the list of "undelivered" messages in the system. Note that messages are an abstraction introduced to model the way objects interact and are "delivered" and reacted to in zero time. Thus some of the states are transitory in that they have zero holding time. Using Stochastic Petri Net terminology, a state is called *vanishing* if it has zero holding time, and *tangible* otherwise. In our system, the vanishing states are those with one or more undelivered messages in the system and the tangible states are those with no undelivered messages in the system.

For the purposes of analysis, we are only interested in transitions between tangible states. We define a vanishing sequence to be a sequence of state transitions that starts and ends in a tangible state where all intermediate states are vanishing. The actions that may happen in response to an event or a message determine sets of vanishing sequences. Once all the sets of vanishing sequences are determined, the transition rates for each pair of tangible states can be easily found. These rates are collected to form the state transition rate matrix which is then fed to the markov chain solver.

The above informal description should provide an intuitive basis on which to interpret the more formal definition which follows. Formally, we say that an object $\mathcal{O}$ is defined as

$$\mathcal{O} \triangleq (I, \mathcal{S}, S_0, \mathcal{E}, \mathcal{M}^r, \mathcal{M}^s, \mathcal{R}, \delta', \delta)$$

where I = The unique name of the object
$\mathcal{S}$ = The set of possible states for the object
$S_0 \in \mathcal{S}$ = The initial state for the object
$\mathcal{E}$ = The set of events that can be generated by the object
$\mathcal{M}^r$ = The set of messages that can be received by the object
$\mathcal{M}^s$ = The set of messages that can be sent by the object
$\mathcal{R}$ = The rate function of the object

$$\mathcal{R} : \mathcal{S} \times \mathcal{E} \rightarrow \Re^+$$

δ' = The event function of the object

$$\delta' : \mathcal{S} \times \mathcal{E} \rightarrow \{(0,1] \times \mathcal{S} \times [\mathcal{M}^s]\}$$

, where $[\mathcal{M}^s]$ is an ordered list of messages each belonging to $\mathcal{M}^s$

δ = The message function of the object

$$\delta : \mathcal{S} \times \mathcal{M}^r \rightarrow \{(0,1] \times \mathcal{S} \times [\mathcal{M}^s]\}$$

δ' is a function that, given a state and an event, returns the set of possible event responses each consisting of a new state, a list of messages to be delivered and the probability that this event response occurs. Associated with the event function, there is a rate function $\mathcal{R}$ that, given a state s and an event e, gives the rate at which event e occurs in state s. δ is analogous to δ' and given a state and a message, returns the set of possible message

responses each consisting of a new state, a list of messages and the probability that this message response occurs.

We say that a system model $\mathcal{X}$ consists of a set of states $\mathcal{S}$, an initial state S_0, and a transition function γ.

$$\mathcal{X} \triangleq (\mathcal{S}, S_0, \gamma)$$

where $\mathcal{S} = \{< S_1, S_2, \ldots, S_N >, [m_1, \ldots, m_n]\}$

N is the number of objects in the model

$S_i \in \mathcal{S}, 1 \leq i \leq N$

$m_j \in \mathcal{M}, 1 \leq j \leq N, \mathcal{M}$ is the set of all messages

$S_0 = < S_{1_0}, S_{2_0}, \ldots, S_{N_0} >$

The system model transition function γ is defined in terms of the object transition functions δ and δ'.

$$\gamma : (< S_1, \ldots, S_i, \ldots, S_N >, [m_1, \ldots, m_k]) \rightarrow (< S_1, \ldots, S_i', \ldots, S_N >, [m_2, \ldots, m_k, m_{k+1}, \ldots, m_{k+n}])$$

iff message m_1 has as a destination object $\mathcal{O}_i$, where $m_{k+1}, \ldots, m_{k+n}$ are messages generated by object $\mathcal{O}_i$ as a response to message m_1, i.e. $(p, S_i', [m_{k+1}, \ldots, m_{k+n}]) \in \delta_i(S_i, m_1), p > 0$

$$\gamma : (< S_1, \ldots, S_i, \ldots, S_N >, []) \rightarrow (< S_1, \ldots, S_i', \ldots, S_N >, [m_1, \ldots, m_k])$$

iff e is an event that $\mathcal{O}_i$ generates and $(p, S_i', [m_1, \ldots, m_k]) \in \delta_i'(S_i, e)$

We define a state $(< S_1, \ldots, S_N >, [m_1, \ldots, m_k])$ to be *reachable* if

$$(< S_1, \ldots, S_N >, [m_1, \ldots, m_k]) \in \gamma^*(S_0) = \gamma^*(< S_{1_0}, \ldots, S_{N_0} >, [])$$

where γ^* is the transitive closure of γ. That is, a state $\mathcal{S}$ is reachable if there is a valid sequence of states from the initial state S_0 to $\mathcal{S}$. A reachable state with no outstanding messages, $(< S_1, \ldots, S_N >), [])$, is said to be tangible.

We define a *vanishing sequence* to be a a sequence of transitions initiated by an event e, $V' \xrightarrow{p'} V_0 \xrightarrow{p_0} \cdots V_K \xrightarrow{p_K} V''$ such that

1. These transitions are allowed by γ and occur with probability p' or $p_i, 0 \leq i \leq K$,

2. V' and V'' are tangible states,

3. $V_0, \ldots, V_K$ are vanishing states.

Let v be a vanishing sequence. Then the transition rate $\alpha(v)$ from tangible state V' to tangible state V'' corresponding to sequence v is given by

$$\alpha(v) = \mathcal{R}(V', e) \times p' \times \prod_{i=0}^{K} p_i$$

where $\mathcal{R}$ is the rate function as defined previously. Let $\mathcal{V}(V', V'')$ be the set of all vanishing sequences from V' to V'', and let $R(V', V'')$ be the transition rate from V' to V''. Then

$$R(V', V'') = \sum_{v \in \mathcal{V}(V', V'')} \alpha(v)$$

The transition rate matrix is $R[V', V'']$ for all reachable tangible states V' and V''.

3 Implementation Organization.

We have developed a prototype written in Prolog based on the object oriented model presented in the previous section. This prototype allows users to easily define, solve and query markov models at a high level of abstraction. This section concentrates on the implementation organization which is largely independent of the use of Prolog. However, since Prolog was perhaps an unusual choice, we will briefly present our reasons for this choice before describing the organization of this prototype.

3.1 Implementation Language

The basic operation in building the transition rate matrix for a model is finding all the reachable states from a given initial state, according to rules that describe the behavior of the model. These rules specify preconditions on the state of an object and the actions to be taken if the preconditions are satisfied. Three features that Prolog provides make it appropriate for this prototype. First, Prolog allows the use of untyped complex data structures which allows the user full freedom in the description of object states. Second, Prolog provides *unification*, a powerful form of pattern matching. The pattern matching is used in the preconditions of the rules to determine which actions can be taken. This allows very general rules to be written quite simply, as will be seen in the examples. Third, Prolog has backtracking search as a basic feature of the language. As more than one rule may have its preconditions satisfied at the same time, the backtracking automatically applies all possible such rules to find all reachable states. These features allow rapid development both of complex models and complex modeling domains.

The problem with using Prolog is that the system is not as fast as custom compiled modeling tools. The flexibility and power more than make up for the slower speed. With better compilers for Prolog becoming available, speed will become even less of an issue.

3.2 Prototype Organization

In order to facilitate the 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.

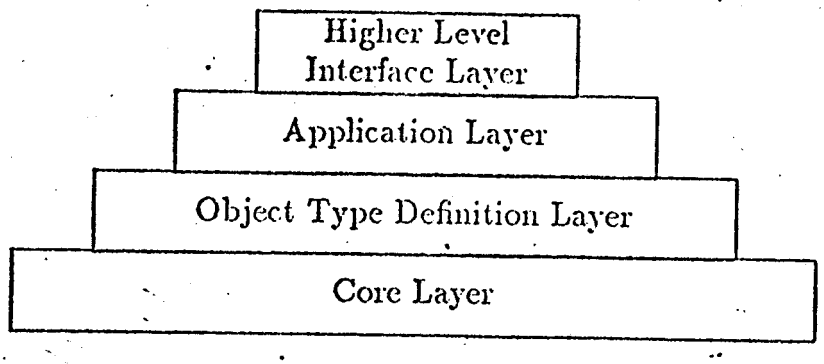

Figure 1: System layers.

3.2.1 Core Layer

The core of the system provides a generic interface which takes descriptions of objects, events, messages 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 expects the following "schema" or format.

1. Each object has a unique name
2. Each object has an initial state
3. Each object has a set of events it can generate, the rates at which they occur, and a list of corresponding actions
4. Each object has a set of messages it can accept and their corresponding actions

The core operates on this description by searching for the states reachable from the given initial state. A standard recursive search procedure can be used that, for each reachable state S, determines the states reachable from S by some event given that the system is in state S.

3.2.2 Object Type Definition Layer

An obvious extension to the basic paradigm presented in the previous section is the notion of object types. The notion of object types is important since it allows economy in specifying a model. It is expected that object type definitions will be parameterized so

that instances of the objects can be declared and parameters specified for each instance. In this section, we introduce by way of example the language used to define object types. The example used is a reliability model consisting of a set of cpus and a set of memories that fail at some rate and a repair service that fixes the cpus and memories at some rate. This is to be modeled as three objects, a cpu object, a memory object and a repair service object. The cpu and memory objects will generate one event, *failure* and will be able to receive one message, *repaired*. The repair_service will generate one event, a *repair*, and will be able to receive one message, *fail*. In this section, the necessary object types will be defined, a component_type for the cpu and memories objects and a repair_type for the repair service object.

```
OBJECT TYPE: component_type
OBJECT NAME: component_name
EVENT failure: N-- > N1
        CONDITION: N > 0
        ACTION: N1 is N - 1, failure_rate(component_name,R),
                send(repair_service,fail(component_name))
RATE: N*R
MESSAGE repaired: N-- > N1
        ACTION: N1 is N + 1

OBJECT TYPE: repair_type
OBJECT NAME: repair_name
EVENT repair: [H|T]-- > T
        ACTION: repair_rate(repair_name,R), send(H,repaired)
RATE: R
MESSAGE fail(Component): L-- > L1
        ACTION: append(L,[Component],L1)
```

This example begins by defining the object type *component_type*. The OBJECT NAME statement gives a formal variable that, when instantiated, will be the name of this object. The OBJECT TYPE statement declares the type of the object being defined. Now the behavior is described. The EVENT statement says that this object can generate a failure event. When it does, the object state at the start of the event is N and the object state at the end of the event is N1. The CONDITION statement gives the conditions under which this event can occur. In this case, a failure can occur when the number of working components, N, is greater than zero. The ACTION statement describes the actions the object takes in response to the event. When there is a failure, the number of working components is decremented and a message is sent to the repair service. The failure_rate(component_name,R) is a request for a parameter which returns the failure rate of the object. Component_name means this object, and R is the return value. If a

response were probablistic instead of deterministic, a PROBABILITY statement would be used. The description for MESSAGE repaired is very similar and is omited.

The OBJECT TYPE repair_type defines the repair service. The major difference between repair_type and component_type objects is that repair_type objects use first-come first-served service discipline so the state is represented as an ordered list rather than a number. When an object is repaired, it is taken off the front of the list. Following Prolog syntax the $[H|T]$ refers to the input state being a list with head H, and tail T, while the $->T$ indicates that the output state is the tail of the input state. When an object is repaired a repair message is sent to it. The append function creates a new list by merging the old list L, with the list containing the name of the newly failed Component.

There are now two objects that are defined that can be used by an end user in building a model. These object types can be combined with others to form libraries. A user can then define instances of object types by simply referring to the library definitions, as will be shown in the following section.

3.2.3 Application Layer

At the application layer, an "end user" can define a model by instantiating objects from an object types library and declaring the required parameters. These required parameters are used to customize individual instances.

```
type(cpu,component_type).
type(memory,component_type).
type(repair_service,repair_type).

initial(cpu,2).
initial(memory,3).
initial(repair_service,[]).

failure_rate(cpu,pfc).
failure_rate(memory,pfm).
repair_rate(cpu,prc).
repair_rate(memory,prm).

value(pfc,0.001).
value(pfm,0.01).
value(prc,0.25).
value(prm,1.0).
```

Continuing with the previous example, the instances of the objects can now be created. The type(cpu, component_type) and type(memory, component_type) statements declare two components to be of object type 'component_type'. Similarly the type(repair_service, repair_type) statement declares the repair_service to be of object type 'repair_type'. The initial(cpu,2), initial(memory,3) and initial(repair_service,[]) statements declare that initially in the model there are two operating cpus, three operating memories and an empty repair queue. It remains only to fill in the rate parameters. The failure_rate(cpu,pfc) gives a symbolic value for the cpu failure rate parameter, similarly with the failure_rate(memory, pfm), repair_rate(cpu,prc), and repair_rate(memory,prm) statements. The state transition rate matrix can be generated at this point in terms of the symbolic parameters. Only when the matrix is actually solved are the numerical values necessary. These values are specified by the value(pfc,0.001), value(pfm,0.01), value(prc,0.25) and value(prm,1.0) statements. The model is complete and ready to evaluate. The results can be examined by a high level query language described in the next section.

3.2.4 Higher Level Interface Layer

The highest layer is where sophisticated user interfaces (e.g. English-like or graphical) can be defined. These would be built on top of the application layer by providing a translation to object types and parameters. We have not concentrated on this layer thus far although we have written a translator for the SAVE [GOYA86] user interface.

3.3 Other Implementation Issues

This section discusses some issues that were omitted from the general description.

3.3.1 Conditions and Functions

There are many times when the behavior of one object is dependent on the state of another. In reliability models, for example, an object can go dormant if certain combinations of components fail. If a disk controller D fails, any disks attached to D become dormant. An object will often have a different failure rate when it is dormant (often zero) than when it isn't. If the disk is modeled as not being able to fail when dormant, the disk object would have to know whether the disk controller object was operational before the disk could fail. To inquire the state of the controller, the disk object could send a message to the controller and change to an intermediate state waiting for a response from the controller. The controller would send back a message indicating whether it was operational or not.

With this information the disk would know whether it could fail. From our experience, it has proven easier to let the disk inquire about the controller object state without resorting to the message mechanism. We allow functions to be declared, as part of an object's specification, that return information about that object's state. The inquiring object uses the evaluate(object, function, Return) predicate, where object is the destination object, function is the service requested from object and Return is the return value. The object responding to the inquiry must have declared the function.

3.3.2 Loops

Similar to Generalized Stochastic Petri Nets (GSPNs), discrete event simulations and other modeling systems that have "zero-time" events, a model can be created that has "infinite" loops of zero-time events.

As an example, consider a queueing system consisting of an infinite server terminals and two cpus with no waiting room. If a customer arrives at one of the cpus and it is busy, the customer is rerouted to the other cpu with probability p_c and to the terminals with probability p_t. Departures from the terminals are routed with equal probability to either cpu. This model has an infinite loop. If both cpus are busy and a customer arrives at one of them, it is routed to the other cpu with probability p_c. The second cpu may reroute it back again with probability p_c (total probability p_c^2). With some probability the customer can keep bouncing from one cpu to the other. More specifically if there are three customers in the system and the system state is represented as the number of customers in the terminals and cpus respectively (plus messages in transit), then the state transitions would be:

$$< 1,1,1 > \rightarrow < 0,1,1 > [cpu_1\ arrival] \xrightarrow{p_c} < 0.1,1 > [cpu_2\ arrival] \xrightarrow{p_c} < 0,1,1 > [cpu_1\ arrival] \xrightarrow{p_c} \cdots$$

As all possible transition sequences must be examined in the direct implementation, an infinite number of sequences have to be examined. This problem can be resolved by noticing that the state of the system when the customer arrives at the first cpu from the terminals is identical to the state of the system when the customer is rerouted from the second cpu back to the first. When an event causes a transition to a vanishing state, a transition probability matrix is constructed containing every state reachable in zero time. This transition probability matrix can be solved to determine the rates between tangible states. See [MARS84] for more details.

3.3.3 Ordering of messages

The order in which messages are delivered can be significant as illustrated in the following example, a reliability model where components can fail and be repared. In addition to failing themselves, the failure of one component may cause another component to fail. The model consists of cpu and disk components where a disk failure causes a cpu to fail. The repair service discipline is first-come first-served. When a disk fails, a message is sent to the cpu to tell it to fail and from the disk to the repair service to request repair. When the cpu receives the fail message, it also sends a message requesting repair. These messages are all sent in zero time. Even though the messages from the cpu and from the disk requesting repair both arrive at the repair service in zero time, if the message from the cpu comes to the repair service first, then the cpu would get repaired first, otherwise the disk would get repaired first. The order of messages determines the order that the components are repaired, even though the messages are nominally delivered at the same time.

In terms of understanding the behavior of a model, this is the weakest point of our paradigm. However, this problem is not unique to our system but rather is shared with every other modeling system with zero time events. If needed, this ambiguity is detectable, but in general proves to be very expensive. To detect ambiguity, all permutations of messages must be tried. If two permutations of messages derive different tangible states, there is an ambiguity. However, many standard types of objects are not affected by the order in which messages arrive, e.g. a processor sharing server is not affected by the order in which customers arrive, and neither is a priority server. In addition, as long as no object that is sensitive to message order receives more than one message in zero time, there can be no ambiguity. This will usually be the case. In those other cases where there is an ambiguity, it should be noted that any ordering of messages can be specified by the modeler.

4 Examples.

In this section we present several examples which further illustrate use of the system. In the first example, we show the ease of using predefined library objects along with user defined objects in a model. The second example is a processor-bus-memory example drawn from the Petri Net literature. The third example is of a more complex condition that could be used in a model. A final example shows an example query of analysis results. These examples show some of the power of our approach.

4.1 Queueing Network Example

In this example we demonstrate the simplicity of building new object types and using them in conjunction with object types from a library. We emphasize the ease with which new objects can be created and used in conjunction with predefined objects. This model is of a simple load balancing system as shown in figure 2. The model is in two parts, the first part instantiating objects from predefined types, and the second describing a new object. The type statements establish the basic objects, a set of terminals using the infinite server queueing discipline, and two cpus using the processor sharing queue discipline. The *initial* statements, *route* statements and *departure_rate* statements fill in the necessary parameters for the objects. The other object needed is the scheduler. Assuming that no such object type exists in the library, the scheduler must be defined specifically for this model.

The scheduler tries to maintain an up to date view of the system by getting new state information from the two cpus. To do this, it samples the queue lengths of the two cpus at intervals which are exponentially distributed with mean $1/ur$. When an arrival comes to the scheduler from the terminals, it is sent to the cpu the scheduler believes has the lowest load. If the scheduler believes the loads are equal, it chooses one cpu or the other with probability 1/2.

The scheduler has a state that is a 2-tuple (n_1,n_2), where

n_i = the queue length last supplied from cpu_i.

The initial state has zero customers in each cpu. The only event that the scheduler generates is the update event, which gets the current state information from both cpus. The scheduler also receives arrival messages from the terminals. Upon receipt of an arrival, the scheduler compares the queue lengths of the two cpus in its most recent load update. If they are the same, one cpu or the other is chosen with probability 1/2, and an arrival

message is sent to that cpu. If they are different, an arrival is sent to the cpu that apparently has the smallest number of customers.

```
type(terminals,inf).
type(cpu1,ps).
type(cpu2,ps).

initial(terminals,3).
initial(cpu1,0).
initial(cpu2,0).

route(terminals,scheduler,1.0).
route(cpu1,terminals,1.0).
route(cpu2,terminals,1.0).

departure_rate(terminals,tr).
departure_rate(cpu1,cr).
departure_rate(cpu2,cr).

OBJECT: scheduler
INITIAL STATE: [0,0]
EVENT update: DONT_CARE-> [N1,N2]
        ACTION: evaluate(cpu1,state,N1),
                evaluate(cpu2,state,N2)
        RATE: ur
MESSAGE terminal_arrival:[M,N]-> [M,N]
        CONDITION: M = N
        ACTION: send(cpu1,arrival)
        PROBABILITY: 1/2
        CONDITION: M = N
        ACTION: send(cpu2,arrival)
        PROBABILITY: 1/2
        CONDITION: M < N
        ACTION: send(cpu1,arrival)
        CONDITION: M > N
        ACTION: send(cpu2,arrival)
```

4.2 Multiprocessor System Model

This next example is from [MARS84] and is a Generalized Stochastic Petri Net (GSPN) model of a multiprocessor system with multiple buses and multiple common memories. We describe this model using our methodology and compare it with the GSPN model. This particular example is a five processor, three common memory, two bus system as in figure 3. The GSPN model for this system is shown in figure 4.

The following description of the system is also from [MARS84] :

> In this model, the processors execute in their private memory for an exponentially distributed random time with mean $1/\lambda$ before issuing an access request directed to one of the common memories in the system. The request may not be immediately served, either because there is no bus available or because the addressed memory is busy. The durations of accessed to common memories are independent, exponentially distributed, random variables with mean $1/\mu$.
>
> The following bus arbitration policy is assumed: if a processor, say A, is in the queue for a memory already accesses by another processor, say B, then at the end of the access the bus is not released but is immediately given to processor A.

Using our methodology, the system is modeled as three objects, CPU representing the cpus, CM the common memories, and B the buses. The CPU object state is just the number of cpus executing in their private memory. The bus state is the number of free buses when positive, the number of customers waiting for buses when negative. The state of the common memory is a three-tuple representing the number of processors waiting for or accessing each memory. Initially all five cpus are executing in their local memory, both buses are free, and no cpu is trying to access a common memory.

The description of the CPU object is very straightforward. When there are $N > 0$ cpus executing in their private memory, they make an access to common memory at a rate of N*lambda. A request to acquire the bus is sent and the state of the cpu object is decremented. The cpu object can receive one message, *process*, the result of a cpu finishing its accessing of a common memory. The state is then incremented as the processor goes into its private memory execution phase.

The bus object implements the arbitration policy. It generates no events but handles two messages, *acquire* and *release*. The handling of the *acquire* message depends on whether there is a free bus, $N > 0$ or not, $N <= 0$. If a bus is free, the request is immediately

granted by decrementing the number of free buses and sending the *access* message to the memories. If no bus is free, the number of free buses is decremented representing the number of common memory access requests waiting for a bus. Handling the *release* message is similar to *acquire*. If there are requests waiting for a bus, $N < 0$, when one is released, the number waiting is decremented and an *access* is sent to the memories.

The memories are described in two different ways. The event, *release* is handled in an initially simpler fashion, explicitly spelling out the transitions. The *access* message is handled in a more general way, but a way that seems more confusing at first. Normally *access* and *release* would both be described in the general way.

The memories object generates a release event at rate mu for each memory being accessed. If there are no more cpus queued for that memory, (N = 1, O = 1, or P = 1), that memory is decremented, the cpu is sent the process message, and the bus is sent the release message. If on the other hand, other cpus are queued for that memory, ($N > 1$, $O > 1$, or $P > 1$), the queue at that memory is still decremented and the cpu is still sent the process message, but the bus is not released.

The *access* message is very similar. It is handled by using a substitute (subst()) that nondeterministically picks one element in the memories state such that it meets the rest of the condition. If there are already cpus queued at the selected memory ($N > 1$), the queue size is incremented and the bus is released. If the memory is free (N = 0), the number of cpus at that memory is set to one. Not only is this more general, it is also more compact.

OBJECT: cpu
INITIAL STATE: 5
EVENT access_common: $N- > N1$
    CONDITION: $N > 0$
    ACTION: N1 is N-1, send(bus,acquire)
    RATE: N * lambda
MESSAGE process: $N- > N1$
    ACTION: N1 is N+1

OBJECT: bus
INITIAL STATE: 2
MESSAGE acquire: $N- > N1$
    CONDITION: $N <= 0$
    ACTION: N1 is N-1
    CONDITION: $N > 0$
    ACTION: N1 is N-1, send(memories,access)
MESSAGE release: $N- > N1$
    CONDITION: $N < 0$

ACTION: N1 is N+1, send(memories,access)
CONDITION: $N >= 0$
ACTION: N1 is N+1

OBJECT: memories
INITIAL STATE: [0,0,0]
EVENT release: [N,O,P] -> [N1,O1,P1]
CONDITION: $N > 1$
ACTION: N1 is N-1, O1=O, P1=P, send(cpu,process)
CONDITION: $O > 1$
ACTION: N1=N, O1 is O-1, P1=P, send(cpu,process)
CONDITION: $P > 1$
ACTION: N1=N, O1=O, P1 is P-1, send(cpu,process)
CONDITION: N = 1
ACTION: N1 is N-1, O1=O, P1=P, send(cpu,process), send(bus,release)
CONDITION: O = 1
ACTION: N1=N, O1 is O-1, P1=P, send(cpu,process), send(bus,release)
CONDITION: P = 1
ACTION: N1=N, O1=O, P1 is P-1, send(cpu,process), send(bus,release)
RATE: mu
MESSAGE access: $M- > M1$
CONDITION: subst(N,M,N1,M1), $N > 0$
ACTION: N1 is N+1, send(bus,release)
CONDITION: subst(N,M,N1,M1), N = 0
ACTION: N1 is N+1

We see two main advantages in our approach over GSPNs. The first is the modularity in describing each object separately and only letting objects interact in one stylized way. This has been discussed earlier. The second advantage is in the generality of the models.

To accommodate an increase in the number of common memories in the processor-bus-memory GSPN model, additional places and transitions must be added. The number of places and transitions needed for this model is linear in the number of memories. As is shown in [MARS84] this linear growth of places and transitions with respect to the number of memories can be avoided by skillfully reorganizing the model. But this is done by making the number of places and transitions linear in the number of buses in the model. In addition, changing from a memory dependent model to a bus dependent one is nontrivial and in our opinion prone to error. In our model, changing the number of bus, cpu or memory components involves only changing the initial state. For example, a ten cpu, six memory, four bus system, would only require changing the cpu initial state from 5 to 10, the bus state from 2 to 4, and the memory state from [0,0,0] to [0,0,0,0,0,0].

4.3 Complex Conditions

Another illustration of the flexibility of our system concerns the manner in which complex conditions and control schemes can be incorporated into a model. For example, in reliability models, a user might have a repair policy that gives priority to any one component that if repaired would make the system operational. If no single component being repaired would make the system operational, the repair policy would be a user defined priority. This kind of behavior can be very difficult to model in other systems because of its ad hoc nature. Note that any one specific example could be incorporated in a system - the real trick is to allow ad hoc rules to be specified. In this section are presented two examples to illustrate these points.

The following two examples are written in Prolog. Prolog gives the user much more expressiveness than the stylized interface language. These predicates written in Prolog can then be used directly by the high level language in the CONDITION statement or ACTION statement.

4.3.1 Data Availability Modeling Example.

This 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 5. 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).

operational(State) :- 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 operational if all the data is available. The operational predicate would typically be used in a condition statement.

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

4.3.2 "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 3 we would add:

trap(State) :- member([repair_facility, [r_1, r_2]], State).

While searching the state space, a transition to a state S for which trap(S) is true, can be automatically changed to a transition to a state *trap_state_name*.

4.4 Querying the Analysis Results

So far, we have concentrated on generating the markov chain. Another feature of our prototype is the query language. If a user is interested in equilibrium state information, she is generally interested in a more compact answer such as the mean queue length at a particular object, or the marginal probability that an object will be in a particular state. To answer this type of query, we allow a user to associate a "reward" with each state. These states are specified by rules similar to "ACTION" rules. To calculate the result, the system simply sums up the product of the state probabilities and the associated rewards.

As an example, consider the load balancing scheduler system model of section 4.1. Suppose we are interested in the percentage of time that the system is in an unbalanced state, i.e., states in which the number of customers at one cpu has at least two customers more than the other cpu. A reward of 1.0 is associated with each state that has this property. This is specified as follows.

QUERY: unbalanced
OBJECT: cpu1
STATE: N1
OBJECT: cpu2
STATE: N2
CONDITION: $N1 - N2 > 1$
REWARD: 1
CONDITION: $N2 - N1 > 1$
REWARD: 1

The actual query would be:

query(unbalanced, Result).

For particular models or libraries, a user might have many queries predefined in a library similar to an object library.

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 the interface was tailored to these application domains makes the system unique among other systems we know of.

The extensibility of the system is easily seen. For 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. As it is a prototype system, we have not concentrate in optimizing its performance. 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.

b. in defining ad hoc models for studies in which the main objective would be to quickly build the model.

References

[BERR82] R. Berry, K.M. Chandy, J. Misra, and D.M. Neuse, "Paws 2.0: Performance Analyst's Workbench Modeling Methodology and User's Manual," *Information Research Associates*, Austin, Texas 1980.

[CARR86] J.A. Carrasco, J. Figueras, "METFAC: Design and Implementation of a Software Toll for Modeling and Evaluation of Complex Fault-Tolerant Computing Systems," *Proceedings of FTCS-16* pp. 424-429, July 1986

[COST81] A. Costes, 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* pp. 72-78, June 1981.

[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*, pp. 84-89, July 1986.

[IRAN71] K.B. Irani and V.L. Wallace, "On Network Linguistics and the Conversational Design of Queueing Networks," *Journal of ACM*, vol. 18, no. 4, pp. 616-629, October 1971.

[WALL72] V.L. Wallace, "Toward an Algebraic Theory of Markovian Networks," *Symposium on Computer-Communications Networks and Teletraffic*, pp. 397-407, April 1972.

[LEND85] P.M. Lenders, "Modeling Distributed Systems with Logic Programming Languages," *Colorado State University, Dept. of Mechanical Engineering*, Ph.D. Dissertation, 1985

[MAKA82] S.V. Makam, and A. Avizienis "ARIES 81: A Reliability and Life-Cycle Evaluation Tool for Fault Tolerant Systems," *Proceedings of FTCS-12* pp. 276-274, June 1982

[MARS84] A.M. Marsan, G. Conte, and G. Balbo, "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessors Systems," *ACM Transactions of Computer Systems*, pp. 93-122, May 1984

[SAUE81] C.H. Sauer, E.A. MacNair, and J.F. Kurose, "Computer Communication System Modeling with the Research Queueing Package Version 2," *IBM T.J. Watson Research Center*, Tech. Rep. RA-128, November 1981

[SAUE84] C.H. Sauer, 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, pp.203-220, January 1984

[STRO87] B. Stroustrup, "What is "Object-Oriented" Programming", *Proceedings of ECOOP*, Paris, June 1987

[TRIV84] K.S. Trivedi, J.B. Dugan, R.R. Geist, and M.K. Smotherman, "Hybrid Reliability Modeling of Fault-Tolerant Computer Systems," *Comput. Elec. Eng.*, Vol. 11, pp. 87-108 1984

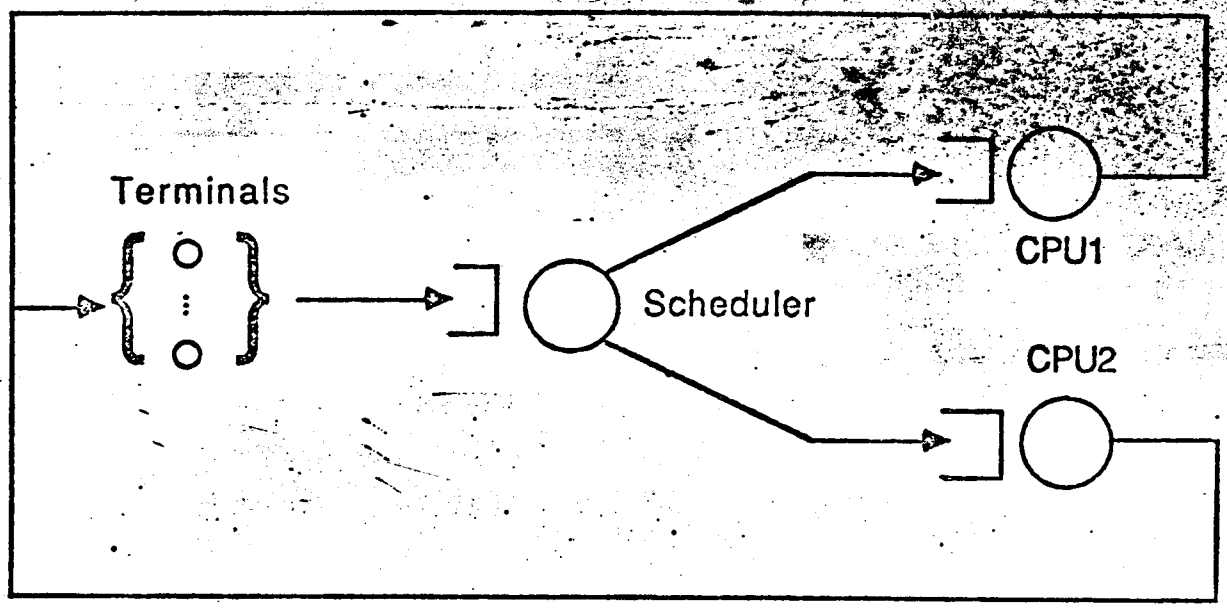

Figure 2 Queueing Network Example

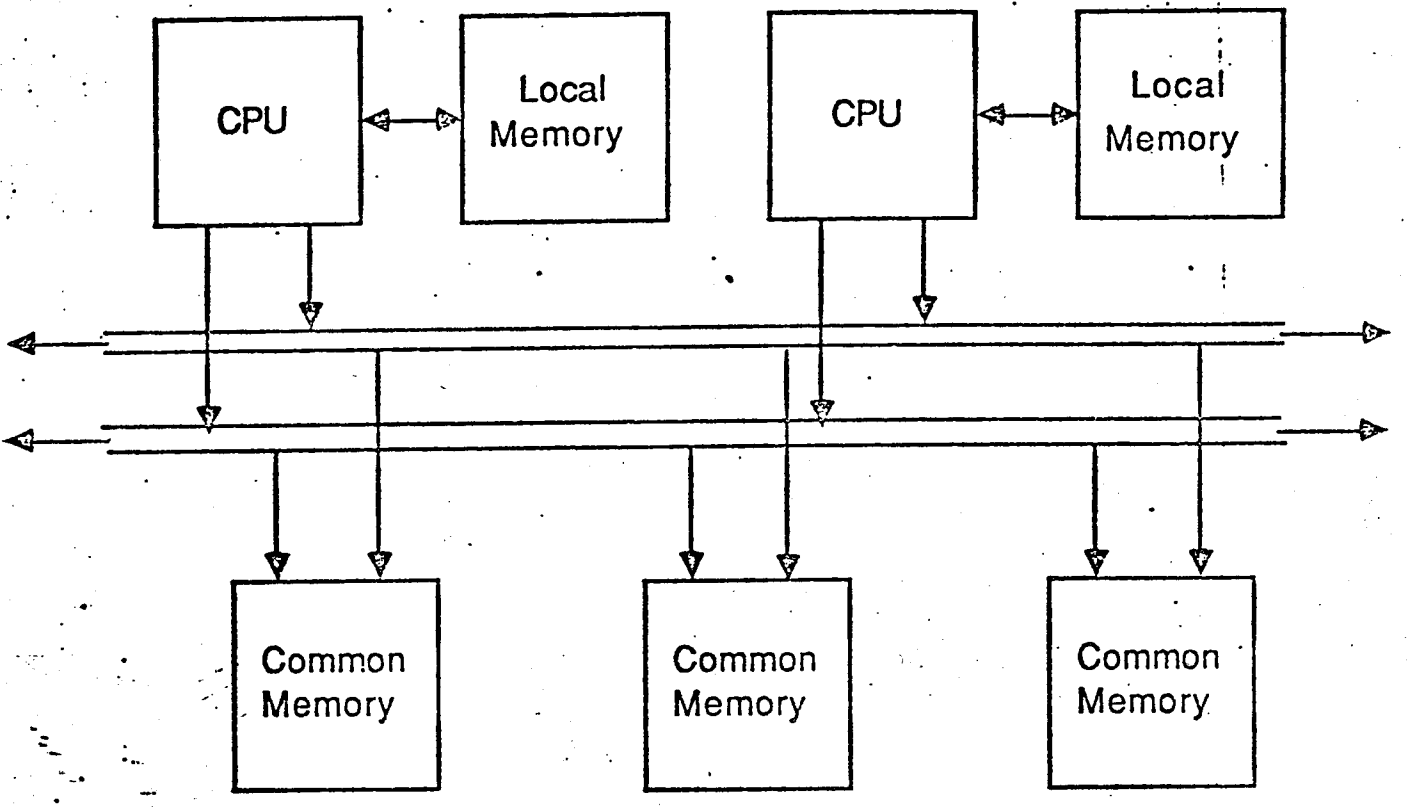

Figure 3 Multiprocessor System Model

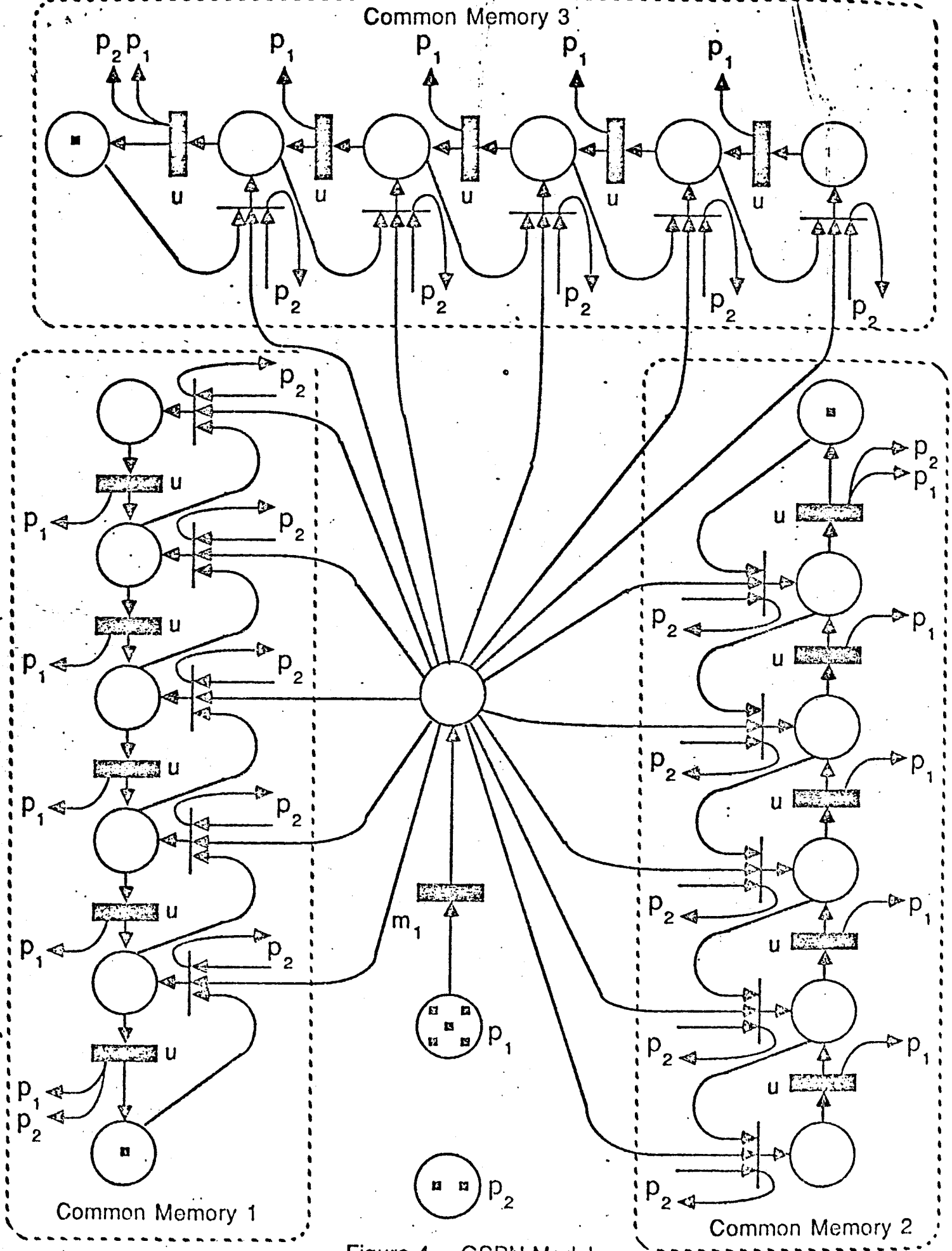

Figure 4 GSPN Model

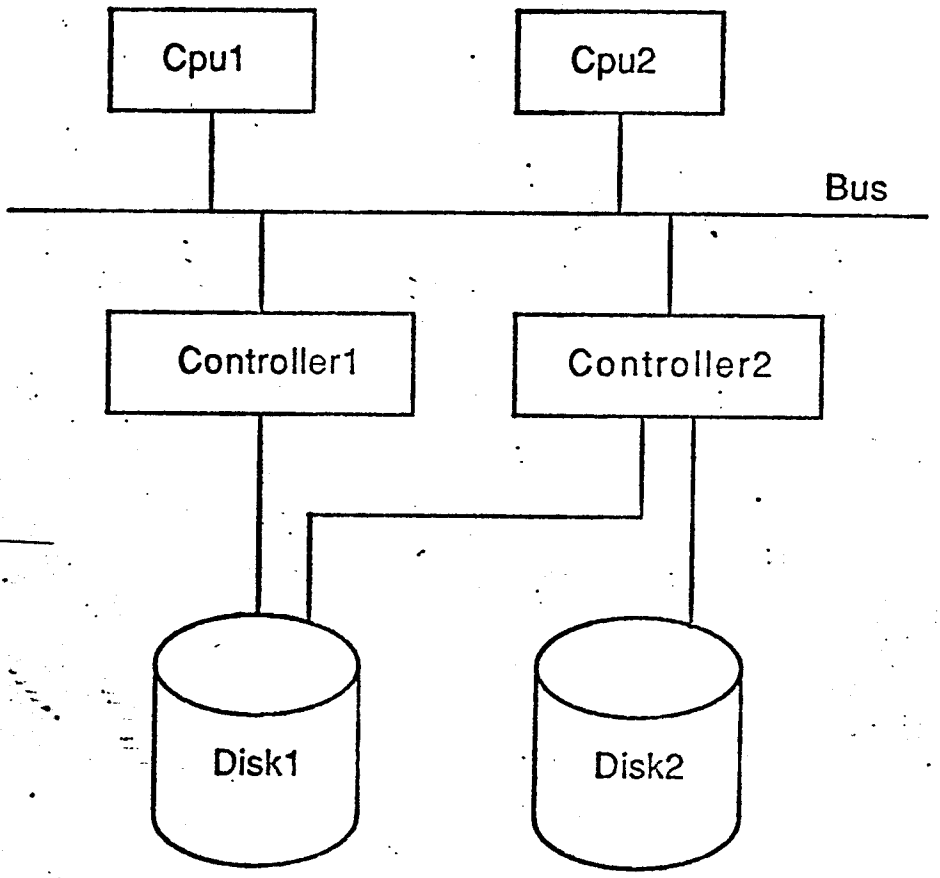

Figure 5 Distributed Architecture Model