A Generalized Timed Petri Net Model for Performance Analysis

MARK A. HOLLIDAY, MEMBER, IEEE, AND MARY K. VERNON, MEMBER, IEEE

Abstract—**We have developed a Generalized Timed Petri Net (GTPN) model for evaluating the performance of computer systems. Our model is a generalization of the TPN model proposed by Zuberek [1] and extended by Razouk and Phelps [2]. In this paper, we define the GTPN model and present how performance estimates are obtained from the GTPN. We demonstrate the use of our automated GTPN analysis techniques on the dining philosophers example. This example violates restrictions made in the earlier TPN models. Finally, we compare the GTPN to the stochastic Petri net (SPN) models. We show that the GTPN model has capabilities for modeling and analyzing parallel systems lacking in existing SPN models. The GTPN provides an efficient, easily used method of obtaining accurate performance estimates for models of computer systems which include both deterministic and geometric holding times.**

Index Terms—**Deterministic delays, dining philosophers, embedded Markov chain, Markov models, performance analysis, Petri nets.**

I. Introduction

We present a Generalized Timed Petri Net (GTPN) model for evaluating the performance of computer systems. The GTPN model is an efficient, easily used tool for calculating exact performance estimates for many models of computer systems. For example, in [3] we derive exact performance estimates of multiprocessor memory and bus interference for fixed memory cycle times. Only approximate solutions existed previously for many of the models we studied.

Petri nets are a graph model of computation [4]. Modifying Petri nets so that time is represented has recently been an active research area. The goal of these models is to analyze system performance as an extension of the reachability analysis. Our model is a generalization of the timed Petri net (TPN) model proposed by Zuberek [1] and extended by Razouk and Phelps [2]. The TPN model associates firing frequencies and deterministic firing times with each transition in the net. Both Zuberek and Razouk and Phelps restrict the allowed nets in order to propose algorithms for building the net's reachability graph. They further restrict the allowed nets in order to analyze the net's reachability graph. Our generalization, the GTPN model, removes all of these restrictions. We present efficient algorithms for building the reachability graph of an arbitrary bounded net and we show how to analyze that reachability graph in the general case.

For the purpose of performance analysis we view the GTPN as a stochastic process. The time-in-state is a deterministic function for each state of the net. However, a probability distribution is defined over the possible next states based on the firing frequencies. The GTPN analyzer automatically generates the associated discrete parameter, embedded Markov chain and calculates performance estimates.

Stochastic Petri nets (SPN's) are an alternate method of representing time in Petri nets for the purpose of performance analysis. Continuous-time SPN models were proposed by Natkin [5], Symons [6], and Molloy [7]–[9], and generalized by Marsan, Balbo, and Conte [10] and Dugan, Trivedi, Geist, and Nicola [11]. The Generalized continuous-time SPN (GSPN) model [10] associates an instantaneous or exponentially distributed firing time with each transition. The Extended SPN (ESPN) model [11] allows general firing time distributions and then defines restrictions on the net which are necessary for efficient solution.

Molloy [7], [9] has also proposed a discrete-time SPN model with transition firing times that are geometrically distributed. The discrete-time SPN model is especially interesting because it can represent deterministic firing times, thus bridging the gap between the SPN and TPN models. However, the representation of deterministic firing times has two restrictions in the discrete-time SPN: 1) a firing time must be a nonzero multiple of some unit time step, and 2) all conflicting actions having deterministic delays must be equally likely.

The GTPN can represent geometric holding times, and thus also bridges the gap between the TPN and SPN models. Furthermore, deterministic firing times can be any nonnegative real value, including zero, and we can assign arbitrary next-state probabilities to conflicting transitions. Thus, the GTPN has some attractive features lacking in existing SPN models.

This paper is an extended version of [12]. In Section II, we describe untimed Petri nets and the previous TPN models. In Section III we describe the GTPN model and in Section IV we show how the GTPN model is analyzed. In Section V, we demonstrate the use of our automated GTPN analysis techniques on the dining philosophers

Manuscript received May 31, 1985. This work was supported in part by the National Science Foundation under Grants DCR-8402680 and MCS81-05904.

M. A. Holliday was with the Department of Computer Sciences, University of Wisconsin—Madison, Madison, WI 53706. He is now with the Department of Computer Science, Duke University, Durham, NC 27706.

M. K. Vernon is with the Department of Computer Sciences, University of Wisconsin—Madison, Madison, WI 53706.

IEEE Log Number 8717402.

model [13], which violates restrictions in the earlier TPN models. In Section VI we compare the GTPN to the discrete-time and continuous-time SPN models. Section VII contains the conclusions of this work and suggestions for future research.

II. Untimed Petri Nets and Previous TPN Models

In this section we describe untimed Petri nets and previous TPN models which provide a foundation for the GTPN in Section III. Section II-A describes untimed Petri nets. A more thorough introduction to untimed Petri nets can be found in Peterson [13]. Section II-B reviews the work of Zuberek and Razouk and Phelps.

A. Untimed Petri Nets

Untimed Petri nets (PN's) contain *places P*, *transitions T*, and *arcs A*. The arcs are directed and can only connect transitions to places and places to transitions. If an arc exists from a place to a transition, then the place is an *input place* for that transition. If an arc exists from a transition to a place, then the place in an *output place* for that transition. Places may contain *tokens*. The *state* of a PN is defined by the number of tokens in each place and is represented by a vector M called the *marking* vector. $M[i]$ is the number of tokens in the ith place.

Petri nets are often illustrated graphically. Circles represent the places. Black dots in the circles represent the tokens. Bars represent the transitions. Fig. 1 can be considered an example of an untimed Petri net by simply ignoring the vectors next to each transition.

The number of arcs connecting a place to a transition is that input place's *multiplicity.* A transition is *enabled* if each of its input places contains at least as many tokens as there are arcs from the place to the transition. The tokens on all input places which exactly equal the input's multiplicity are the transition's *enabling* tokens. An enabled transition can *fire*. A transition *fires* by: 1) removing all of its enabling tokens from its input places, and 2) placing on each of its output places one token for each arc from the transition to that output place. Each firing of a transition changes the assignment of tokens to places and thus creates a new state. The *reachability set* of a PN and a given initial state is the set of all states that can be reached from that initial state via a sequence of transition firings. The *reachability graph* associated with a reachability set can be constructed as follows. Represent each state by a vertex and place a directed edge from vertex v_1 to vertex v_2 if the state v_2 can result from firing some transition enabled in state v_1.

B. Previous TPN Models

Ramachandani [14] was the first to introduce a fixed firing time with each transition in a Petri net. Ramamoorthy and Ho [15], Zuberek [1], and Razouk and Phelps [2] are three more recent studies that use a single fixed firing time. Our work is based on the TPN model of Zuberek and Razouk and Phelps.

The TPN model [1], [2], is a Petri net which has been augmented to include a set of firing durations (D), a set of firing frequencies (F), and a set of named resources (R). Each set is associated with the transitions in the net. Letting S denote the set of reachable states, $\mathcal{R}^+$ denote the positive reals, and $\mathcal{P}$ denote the power set, the model is formally defined as follows:

$$\text{TPN} = (P, T, A, M_0, D, F, R)$$

where

$$\begin{array}{ll}
P = \{p_1, p_2, \dots, p_n\} & \text{(places)} \\
T = \{t_1, t_2, \dots, t_m\} & \text{(transition)} \\
A\colon \{P \times T\} \cup \{T \times P\} \to \{0, 1, 2, \dots\} & \text{(directed arcs)} \\
M_0\colon P \to \{0, 1, \cdots\} & \text{(initial marketing)} \\
D\colon T \times S \to \mathcal{R}^+ \cup \{0\} & \text{(firing durations)} \\
F\colon T \times S \to \mathcal{R}^+ \cup \{0\} & \text{(firing frequencies)} \\
R\colon P \cup T \to \mathcal{P}(\{r_1, r_2, \dots, r_k\}) & \text{(resources)}
\end{array}$$

The state of a TPN is defined differently than in untimed Petri nets because firing a transition is not an atomic operation. A transition has an associated deterministic firing duration. There is a *start firing*, and an *end firing* event. In between the firing is *in progress*. The removal of tokens from a transition's input places occurs at start firing. The placement of tokens on a transition's output places occurs at end firing. While the firing of a transition is in progress, the time to end firing, called the *remaining firing time* (RFT), decreases from the firing duration to zero (without causing a change in the net). Because firings can be in progress when a marking change occurs, a state is only partially defined by the distribution of tokens. A state must also include the RFT of each firing in progress. A state is thus a marking vector and a set of RFT's.

Also unlike an untimed Petri net, the next state is not generated by a single start firing or end firing event. Instead it is generated by a *set* of start firings or a *set* of end firings which occur simultaneously. Given a particular state, the basic rule for finding the possible next states is straightforward. Find how many *enablings* of each transition exist. (Instead of a transition being either enabled or not, it has a nonnegative number of *enablings*. N enablings of a transition exist if each of its input places contains a number of tokens equal to at least N times its multiplicity.) Find the *maximal* sets[1] that can start firing simultaneously. Each maximal defines a next state. The time spent in the original state is zero. The RFT vectors

[1]A set with property α is a *maximal* set with property α if it is not a proper subset of any other set with property α.

of the transitions which just started firing are set to their transition's duration. The frequencies are used in assigning probabilities to next states formed by the start firings of maximal sets. If there are no enablings, but there are some firings in progress, then the next state is generated by the end firing of all transitions with the smallest RFT (Tmin). The time-in-state value in this case is Tmin. If there are no enablings and no firings in progress, then the net remains in the current state forever.

The rule that next states are generated by sets of events that occur simultaneously, is not strictly necessary. The advantage of having it is that the state spaces generated are dramatically smaller. The disadvantage is that the algorithms for building the state space are more complicated.

Zuberek suggested that the reachability graph of the timed Petri net be viewed as a Markov chain and that performance measures be computed using standard techniques for analyzing the Markov chain's long run behavior. Extensions of his work, however, are desirable in two areas. One, he only proposed a method for constructing a net's reachability graph for a restricted class of nets: nets that are *safe* and *free choice*. A net with a given initial state is said to be *safe* if, for every state in the reachability set, no place has more than one token. A net is *free choice* if each place that is an input to more than one transition is the only input to those transitions. Two, even for safe and free choice nets, the structure of the reachability graph (i.e., the Markov chain) may be such that Zuberek's approach gives incorrect values for the performance measures. The state in a discrete time Markov chain can be divided into classes. A set of classes, called *recurrent* classes, is important because in the long run the model will reach and stay in one of these classes. Zuberek's approach gives correct values only when there is exactly one recurrent class.

Razouk and Phelps [2] extend Zuberek's work in the first of the two areas above. They allow a superset of the class of safe, free choice nets. Two or more transitions are said to be in the same *conflict set* if their sets of input places intersect. Two conflict sets *overlap* if at least one transition is in both. Razouk and Phelps make the restrictions that conflict sets do not overlap and that all transitions in a conflict set are *mutually disabling*, i.e., firing of one, disables all the others. They maintain Zuberek's restriction in the second area (they call this, requiring a *cyclic* net).

Razouk and Phelps also introduce the concept *resources*, originally proposed in E-Nets [16]. A resource in their model can be associated with one or more transitions. Whenever one of those transitions is firing, the resource is in *use*. If more than one of these transitions is firing simultaneously, the resource has several *usages* occurring. By building and analyzing the net's reachability graph we can find the average number of uses of a resource over time. This average, if properly implemented and interpreted, can be used to obtain a variety of meaningful performance estimates.

III. The GTPN Model

The GTPN model extends the models of Zuberek and Razouk and Phelps by: 1) removing all restrictions on the net except the obvious one that the state space be finite, and 2) computing correct performance estimates for any reachability graph (i.e., an arbitrary embedded discrete parameter, finite state Markov chain). We also allow the firing duration to be an arbitrary real number (the noninteger case is not discussed by Zuberek or Razouk and Phelps) and we allow resources to be associated with places as well as transitions.

A third extension we have found useful involves firing durations and frequencies. In the models of Zuberek and Razouk and Phelps, the duration and frequency are state-independent constants. In the GTPN model a transition's firing duration and frequency can be expressions containing immediate values (real and integer), names of places, names of transitions, and arithmetic, relational, and logical operators. A place name stands for the number of tokens in that place in the current state. A transition name represents the value one if at least one firing of that transition is in progress in the current state, and is otherwise zero. The state-dependent durations and frequencies become deterministic values when used to determine time-in-state and next state probabilities for a state in the reachability graph. (Note that Petri net *inhibitor arcs* can be modeled using the state-dependent frequency expressions.)

Besides a firing duration, frequency, and set of resources, a GTPN transition has a flag associated with it that is used in computing the next state probabilities as described in Section III-B.

Fig. 1 shows an example GTPN net, including the initial state distribution of tokens. Each place and transition is labeled. Each transition has, from left to right, its firing duration expression, its frequency expression, its flag, and its list of resources. This example models users at terminals, who with a geometric think time generate requests for a server. There is one token on place P1 for each user. Transitions T1 and T2 implement the think time. Transition T3 implements a load-dependent server with a firing duration that depends on the number of tokens on P2.

In Fig. 2 and Table I we show the reachability graph for the simple GTPN in Fig. 1 assuming only one user. The labels on the edges of the graph are the next state probabilities. The labels on the vertices of the graph are the values for time-in-state. The marking vectors are shown in the table. The RFT sets are shown as a list of pairs, with one pair per in-progress firing of a transition. The first component of each pair is the name of the transition. The second component is the remaining firing time. The resources used and their number of uses are also shown in the table.

The Razouk and Phelps' TPN model does not allow multiple tokens on place P1. Allowing such nets complicates constructing the reachability graph. An overview of our reachability algorithm, which handles these compli-

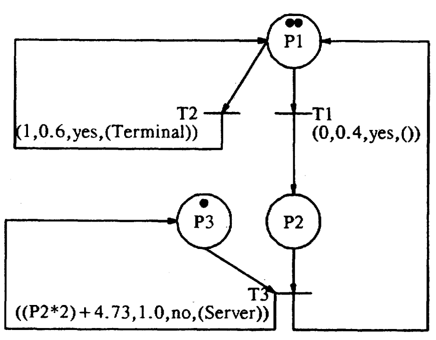


Fig. 1. Example of a GTPN net.

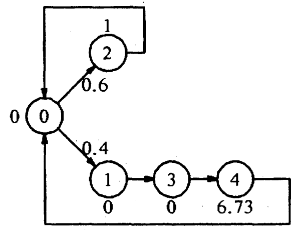


Fig. 2. Reachability graph for example.

TABLE I
REACHABLE STATES FOR EXAMPLE

States	Marking			RFT Set	Resources
	P1	P2	P3		
0	1	0	1	{}	{}
1	0	0	1	{(T1,0.0)}	{}
2	0	0	1	{(T2,1.0)}	{Terminal(1)}
3	0	1	1	{}	{}
4	0	0	0	{(T3,6.73)}	{Server(1)}

```
X.State ← Initial State; X.Class ← Frontier
while at least one Frontier state, Y do begin
    if Y is a duplicate of an Interior state Z then
        Y.Class ← Duplicate
    else begin
        Find the set of enablings in Y
        If no enablings and the RFT set is empty then
            Y.Class ← Terminal
        else if any enablings then begin
            Find the set of maximals of enablings
            Compute the probability of each maximal
            For each maximal M create a new state Z from Y
                Remove tokens from the input places of
                    transitions that have enablings in M
                Add a firing, $f_t$, to the RFT set for each
                    enabling in M
                Set the RFT of each added firing, $f_t$, to the
                    firing duration of transition t
                Z becomes a child of Y; Z. Class ← Frontier
            for all resources ResUsages[Y] ← count uses
            TimeInState[Y] ← 0; Y.Class ← Interior end
        else begin
            Let Tmin be the smallest RFT in Y
            Create state Z from Y by subtracting Tmin from
                each RFT in Y
            For each firing $f_t$ whose RFT = 0 in Z do
                Add tokens to the output places of transition t
                Remove $f_t$ from the RFT set
            Z becomes a child of Y; TimeInState[Y] ← Tmin
            for all resources ResUsages[Y] ← count uses
            Z.Class ← Frontier; Y.Class ← Interior end
end
```

Fig. 3. Overall state space algorithm.

cations, is in Fig. 3. The TimeInState and ResUsages (number of usages of a resource) functions are used in the performance analysis as described in Section IV. The algorithm has two complex parts: 1) finding the next states when the next states are due to maximal sets of transition enablings which start firing together, and 2) assigning probabilities to next states. These two parts are discussed in Sections III-A and III-B.

A. Finding Maximals

Our first point is somewhat discouraging. In an arbitrary state in a net the number of maximals in the worst case is exponential in the number of enablings.

Theorem 1: Consider a state S in a GTPN with n enablings. The number of maximals is

$$\Omega\left(2 \cdot \frac{2^n}{\sqrt{2\pi n}}\right).$$

Proof: First we will construct a state in a net such that the number of maximals is $\binom{n}{n/2}$. Consider the case where n is even (the n is odd case is similar). Consider the net with n transitions and one place P which is an input place for all the transitions. Assume each of the n transitions also has another input place, with one initial token, which is also an output place of the transition. Let the place P have $n/2$ tokens. In this state, there are n enablings (i.e., each transition is enabled once), and $\binom{n}{n/2}$ maximals. Thus, the number of maximals is $\Omega\binom{n}{n/2}$. The observation that by Stirling's approximation, $n! = \sqrt{2\pi n}\,(n/e)^n\,(1 + O(1/n))$, completes the proof. ■

This result should not be given too much weight. In practice, we find that the number of maximals is far less than exponential in the number of enablings. Theorem 1 does, however, point out that the space of potential maximals is large. Consequently, an algorithm for finding the maximals must be carefully thought out in order to prevent poor performance when the actual number of maximals is small. The algorithm described below meets this criterion. When we profiled our GTPN tool, the percentage of total program time taken by this algorithm was less than 5 percent for the analysis of large nets.[2]

Our algorithm consists of two independent subalgorithms *Partition* and *FindMax*. The Partition algorithm partitions the set of enablings into *Generalized Conflict Sets*, such that maximals of the partitioned sets can be efficiently combined to generate all the maximals for the original set of enablings. The Partition algorithm does not specify how the maximals for each partition member are found. Maximals are found for each member of the partition by FindMax.

1) The Partition Algorithm: The Partition algorithm has two parts. The first part constructs a static partition of the set of transitions T. Define the *directly-conflicts-*

[2]We profiled our program using *gprof* under 4.2 bsd UNIX® running on a VAX-11/780®.

with relation on T as follows. For all t_1 and t_2 in T, t_1 directly-conflicts-with t_2 if the set of input places for t_1 intersects the set of input places for t_2. Define the *conflicts-with* relation on T as the transitive closure of the directly-conflicts-with relation. The conflicts-with relation is clearly reflexive, symmetric, and transitive, so it is an equivalence relation on T. The partition of T induced by conflicts-with is denoted by $\{GCS[i] \mid 1 \le i \le N\}$, where $GCS[i]$ is the ith member of the partition and N is the size of the partition (GCS stands for generalized conflict set). Note that a transition which does not share any input places with any other transitions forms a GCS of size one.

Part two of the Partition algorithm uses the conflicts-with partitioning of T to partition the set of enablings, *Enablings* (S), for each state S. The desired partition of *Enablings* (S) is

$$\{EGCS[S, i] = Enablings(S) \cap GCS[i] \mid \text{partition } i\}.$$

At this point, FindMax is applied to each partition member to find its *local maximals*. Let *Maximals* [*Enablings*(S)] be the set of maximals over all the enablings. We have constructed our partition such that *Maximals* [*Enablings* (S)] is simply the Cartesian product of the local maximals.

$$Maximals[Enablings(S)] = FindMax(S, EGCS[1]) \times \cdots \times FindMax(S, EGCS[N])$$

2) The FindMax Algorithm: We want to find the local maximals for a generalized conflict set G in a given state S. Any subset of $EGCS[S, G]$ is a potential local maximal (for brevity's sake we will call a local maximal, a maximal, in this section). The power set, $\mathcal{P}(EGCS[S, G])$, unfortunately, can be a large search space. Our goal is to minimize the number of members of this power set that we examine. Note that set inclusion defines a partial order on $\mathcal{P}(EGCS[S, G])$, which, in turn, induces a directed acyclic graph [see Fig. 4(a)]. This graph has one root which represents the set $EGCS[S, G]$ itself. The FindMax algorithm does a breadth-first search of this graph searching for vertices that are maximals. The traversal is implemented in the standard way using a queue. Initially, the root vertex is the only entry on the queue.

Searching the graph breadth-first causes the order in which vertices are examined to have an important property: if 1) the set of enablings E_1 represented by a vertex can fire together, and 2) E_1 is not a subset of any maximal already found, then E_1 is a maximal. In other words, it is impossible that some vertex examined later will have a set of enablings E_2 that can all fire together and for which E_1 is a subset. Thus, to find the maximals we just do the breadth-first search, checking each vertex to see if it satisfies properties 1) and 2).

Three methods are used to avoid searching the entire graph. One, if a vertex's set of enablings, E_1, can all fire together, then none of its descendents needs to be examined (so its children are not added to the queue). Two, a pointer is used to ensure that vertices are never examined more than once. Note that the graph is not a tree. Thus, a naive breadth-first search would cause vertices to be examined multiple times. Three, the pointer used in method two is used to implement a heuristic for pruning subtrees.

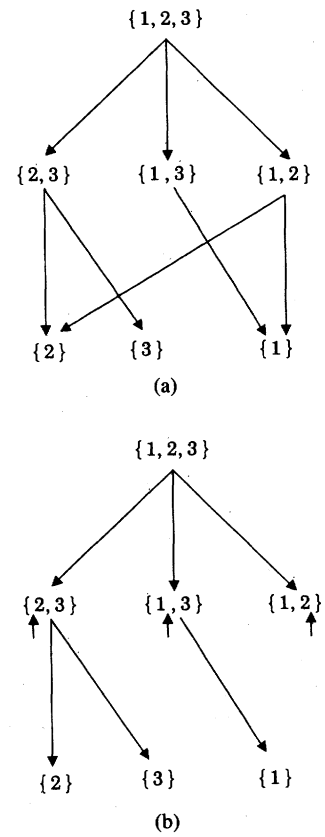


Fig. 4. Example of the FindMax data structures. (a) Directed acyclic graph. (b) Optimized search.

Methods two and three are based on a pointer V_p associated with each vertex V in the graph. Consider the set of enablings E_1 represented by V to be described by a vector of nonnegative integers. The ith component of the vector gives the number of enablings of the ith transition. V_p points at one of the components in this vector. The set of transitions to the left of V_p is V_L. The set of transitions to the right of V_p and including the transition pointed to by V_p is V_R.

The pointer V_p can be used to ensure that a vertex is only examined once. The method is as follows. Set the V_p of the root vertex to point to the leftmost transition. For each parent vertex, subtracting one enabling from the transitions in V_R defines a child vertex. For each of these child vertices, set its V_p to point to the transition that was decremented. It can be shown inductively that at each level in the graph, the sets V_L are distinct for each vertex. Consequently, any child resulting from decrementing V_R of one parent can never be the same as a child resulting from decrementing V_R of another parent. Note, also, that the use of V_p does not cause any state to not be examined that should be examined. For each state c that should be examined, there is a vertex p whose set of enablings over the transitions in V_L matches c's set of enablings over the same transitions. Consequently, c will become a child of p.

Method three is a simple optimization made possible by the existence of V_p. If the enablings in V_L cannot fire together, then it is irrelevant what enablings are subtracted from V_R. Consequently, none of the child vertices resulting from decrementing V_R need be added to the queue.

Fig. 4 illustrates the FindMax algorithm. The left graph is the complete directed acyclic graph defined on the power set of $\{1, 2, 3\}$ by set inclusion. The right DAG shows only the edges that will be used in the breadth first search. The vertical arrows in the middle row are the pointers used in the optimizations.

Partition and FindMax are used to calculate all maximal sets of enabled transitions which can start firing together. Next state transition probabilities must be assigned to these maximal sets. This is discussed in the next section.

B. Computing Probabilities

To interpret the reachability graph as a Markov chain we need to assign probabilities to next states. In the nontrivial case, we need to assign a probability to each maximal set of transitions that can start firing together. From the previous section we know that a maximal is the union of a set of *independent* local maximals, one from each GCS. Thus a reasonable probability for the maximal is the product of the probabilities for the local maximals. Suppose that *LocalMax*$[k, i]$ is the kth local maximal of the ith generalized conflict set. Suppose that there are N generalized conflict sets and the jth global maximal is the union over all GCS's of the j_ith local maximal of the ith generalized conflict set. Then

$$Pr\{Maximal[j]\} = \prod_{\{i: i=1,\cdots,N\}} Pr\{LocalMax[j_i, i]\}$$

In order to compute the probability of a local maximal, we take the product of the frequencies of all the enablings in the local maximal. This is multiplied by a number *NumComb* discussed below. Then for each local maximal, we normalize this product by dividing it by the sum of the products over all local maximals. More formally, suppose the ith GCS has M local maximals and the frequency expression for the kth enabling in the ith GCS is f_k. Then our formula for $Pr\{LocalMax[j_i, i]\}$ is:

NumComb means *number of combinations* and is a combinatoric value associated with each local maximal. This value is defined as the number of ways tokens can be removed from input places in order to implement that local maximal. As a simple example, suppose the local maximal consists of one enabling of one transition with one input place, two arcs connect the place to the transition, and the input place has three tokens. In this case, the combinatoric value *NumComb* is $\binom{3}{2}$.

Computing *NumComb*[*LocalMax*] is done by decomposing it first on the transitions in the local maximal and second on the input places for the given transition. The number of ways that the tokens can be removed from the given input place by the given transition is a binomial coefficient. As we consider each transition, the value in the upper position of this binomial coefficient is changed to reflect any removals made by transitions considered earlier.

More formally, consider the local maximal *LMax* of the ith generalized conflict set. Let *InputPlace*$[t]$ be the set of input places for transition t. Let *Enab*$[t]$ be the number of enablings of transition t in this maximal. Let *TokNeeded*$[t, p]$ be the number of tokens needed from input place p by one enabling of transition t. Note that *Enab*$[t]$ $*$ *TokNeeded*$[t, p]$ is the number of tokens needed from place p by transition t in this maximal. Let *TokLeft*$[p]$ initially be the number of tokens on input place p in the parent state. After looking at an input place p, *TokLeft*$[p]$ is updated to reflect the start firing of all the enablings of transition t. With this notation we have:

$$NumComb[LMax] = \prod_{\{t: t\in LMax\}} \prod_{\{p: p\in InputPlaces[t]\}} \cdot \binom{TokLeft[t, p]}{Enab[t] * TokNeeded[t, p]}$$

From practical experience it appears that in some cases it is reasonable to use this combinatoric value when assigning probabilities to local maximals. A Boolean flag associated with each transition specifies whether this should be done. Only if the flag is *yes* for all transitions in the maximal, is *NumComb* used. Note that if NumComb is used, the next state probabilities are the same as if we constructed the reachability graph by allowing one start firing event at a time with all CntComb flags set to zero, and then summed the probabilities over all paths leading to the state which represents the maximal set.

$$Pr\{LocalMax[j_i, i]\} = \begin{cases} \dfrac{NumComb[LocalMax[j_i, i]] \times \prod_{\{k: k\in LocalMax[j_i,i]\}} f_k}{\sum_{\{m: m=1,\cdots,M\}} NumComb[LocalMax[m, i]] * \prod_{\{k: k\in LocalMax[m,i]\}} f_k}, & \\ \text{if CountComb} = \text{yes}; & \\ \dfrac{\prod_{\{k: k\in LocalMax[j_i,i]\}} f_k}{\sum_{\{m: m=1,\cdots,M\}} \prod_{\{k: k\in LocalMax[m,i]\}} f_k}, & \\ \text{otherwise.} & \end{cases}$$

The motivation for our method of assigning probabilities to local maximals is that it assigns the right probabilities in the important case where all the enablings in a local maximal are independent events. In the case where there are dependencies between the enablings it is difficult to envision a single formula that will always generate the "right" probabilities. This case motivated our introduction of state-dependent frequency expressions. Such frequency expressions can specify what probabilities maximals should have in different markings. Allowing state-dependent frequency expressions also allows the possibility that in some states a transition's frequency expression may evaluate to zero even though it has one or more enablings. We remove these enablings from the set of considered enablings before finding the local maximals.

IV. GTPN Performance Analysis

For the purpose of performance analysis, we view the GTPN as a stochastic process. The time-in-state is a deterministic function, TimeInState, of the state. Nevertheless, the process is stochastic because of the probability distribution over the possible next states. Since the time-in-state can be an arbitrary real number, the process is a continuous time stochastic process. The parameter set is described in Section IV-E. The states of the stochastic process are divided into classes. In the long run, with probability one, the process will reach and stay in one of the set of classes called *recurrent* classes.[3] Consequently, the *long run fraction of time spent in each state* depends on which recurrent class the process reaches in the long run. For each recurrent class the long run fraction of time spent in each state forms a probability distribution over the states. Thus, there is a vector of long run probability distributions with one component for each recurrent class. In addition, we can compute the probability (the *absorption* probability) of reaching each recurrent class in the long run. These absorption probabilities allow us to assign relative weights to the components of the vector of long run probability distributions.

The number of usages of a given resource is also deterministic for a given state; it is a function ResUsages of the state. Consequently, ResUsages, being a function of a random variable, is a random variable. A performance estimate for a resource is a vector with one component for each recurrent class. The value for recurrent class R's vector component is the long run expectation and distribution of that resource's ResUsages random variable, with respect to R's long run probability distribution. In other words, the expectation is the weighted sum of the long run fractions of time spent in each state, given that in the long run the process is in class R. The weight of a state is the number of resource usages in that state. The distribution of a ResUsages random variable is obtained by summing the probabilities of the states that use the resource the same number of times, given that in the long run the process is in class R.

[3]Note that a terminal state in the reachability graph is a recurrent class due to the self-loop we added (see Fig. 3).

Our approach to computing performance estimates uses the key observation[4] that the times at which state changes occur form an embedded, discrete time, finite state Markov chain. Consequently, our approach has four parts: 1) building the Markov chain, 2) aggregating the states in order to reduce the size of the state space, 3) analyzing the Markov chain, and 4) computing resource usage distributions and expectations in the original stochastic process.

Building the Markov chain involves building the reachability graph and assigning next state probabilities, as described in Section III. Our aggregation rule is: any state S_2 can be aggregated with its parent state S_1 if and only if S_1 is S_2's only parent and S_2 is S_1's only child. The number of usages of a resource in an aggregated state equals the sum of its usages in the internal states weighted by the relative fraction of time spent in each internal state. Part 3, the Markov chain analysis, has three steps: a) finding the chain's recurrent classes, b) finding the absorption probability for each recurrent class, and c) finding, for each recurrent class, the long run fraction of visits to each state. These steps are discussed, respectively, in Sections IV-A, IV-B, and IV-C.

Part 4 has two steps: a) for each recurrent class R, computing the long run fraction of time spent in each state (the TimeInState function and the long run fraction of visits to each state are used to do this), b) for each recurrent class R, use the ResUsages functions and the long run fraction of time spent in each state to find the long run distribution and expected number of usages of each resource. These two steps are discussed in Section IV-D.

As mentioned above, the long run resource usage distributions and expectations are the performance estimates (given that the process is in class R in the long run). For each resource we thus have a vector of performance estimates. If desired, the expectations could be weighted by the absorption probabilities to give a single performance estimate.

In Section IV-E we give a more precise definition of the parameter set of the GTPN stochastic process.

A. Finding Recurrent Classes

In order to find the recurrent classes we need to first define a *recurrent* class. Recall that $\forall n \; P_{ij} = Pr\{X_{n+1} = j \mid X_n = i\}$. $P = [P_{ij}]$ is the *one-step transition probability matrix*. $P^{(n)}$, the *n-step transition probability matrix* is defined similarly. $f_{ii}^{(n)}$ is the probability that, starting from state i, the first return to state i occurs at the nth transition. A state is *recurrent* if $\Sigma_{n=0}^{\infty} f_{ii}^{(n)} = 1$. In other words, a state is recurrent if and only if, after the process starts from state i, with probability one the process returns to state i in a finite length of time. A state is *transient* if it is not recurrent. State j is said to be *accessible* from state i if $P_{ij}^{(n)} > 0$ for some integer $n \geq 0$. Two states i and j that are each accessible to the other, are said to *communicate*.

[4]The GSPN model uses a similar observation.

Note the following facts. *Accessibility* defines a partial order on the states. This partial order implies a directed graph with the vertices being the states. *Communication* is an equivalence relation on the states. Thus it partitions the states into subsets called *classes*. The *communication* classes are the strongly connected components of the accessibility graph. These strongly connected components form a directed acyclic graph, DAG.

All the states in a class are recurrent or none are, so we can speak of *recurrent* classes and *transient* classes. In the case of a finite state space,[5] the *recurrent* classes are the leaves of the DAG[17] of strongly connected components. The interior nodes of the DAG are the *transient* classes.

Given that the recurrent classes are the leaves of this DAG, the algorithm to find the recurrent classes is immediate. The reachability graph of the GTPN is the accessibility graph. Create the DAG by finding the strongly connected components of the accessibility graph. We do this using Tarjan's $O(n)$ algorithm [18]. Then find the leaves of the DAG by a depth first search starting at the initial state.

B. Absorption Probabilities

In a finite state Markov chain, if we start in a state in a transient class we will eventually reach and stay forever in one of the recurrent classes. We are said to be *absorbed* by a particular recurrent class. Computing the probability of absorption in a particular recurrent class R given a particular initial state i can be done using a standard technique called *first-step analysis* [19]. On the first state change, the process will move from state i to a state j that is in a transient class or in a recurrent class. If j is in class R, the future probability of being absorbed by class R is one. If j is in another recurrent class, the future probability of being absorbed by class R is zero. If j is in a transient class, then, by the memoryless property, the probability of being absorbed by class R is the same as if j were the initial state.

More formally, suppose that the states in all the transient classes are numbered $0, \cdots, r-1$ and consider a fixed recurrent class R and fixed initial state i. Let $U_i = Pr\{\text{Absorption in class } R \mid X_0 = i\}$ for $0 \le i < r$.

$$U_i = \sum_{\{j \in R\}} P_{ij} + \sum_{j=0}^{r-1} P_{ij} U_{jR}, \qquad i = 0, 1, \ldots, r-1.$$

This equation cannot be solved in isolation. However, if we consider all possible initial states, then we have a system of linear equations that can be solved for the U_i's. The *mean time to absorption* can be computed using a similar system of linear equations.

C. Long Run Expected Fraction of Visits

If R is a recurrent class in a finite state space, then $\forall j \in R$ a number π_j exists such that $\forall i \in R$

$$\lim_{m \to \infty} E\left[\frac{1}{m} \sum_{k=0}^{m-1} 1_{\{X_k = j\}} \,\Big|\, X_0 = i\right] = \lim_{m \to \infty} \frac{1}{m} \sum_{k=0}^{m-1} P_{ij}^{(k)} = \pi_j.$$

The leftmost expression above is *the long run expected fraction of visits spent in state* j. $1_{\{X_k = j\}}$ is the indicator random variable that equals one when the outcome chosen is in the event $\{X_k = j\}$ and 0 otherwise.

We want to find these π_j's for each class R. We do this by noting that each recurrent class R in a finite state space has one and only one *stationary probability distribution* and the vector π_R of its π_j's is this stationary probability distribution. This stationary probability distribution is easy to find since it is the unique solution to the set of equations

$$\pi_R = \pi_R P_R \quad \text{and} \quad \sum_{j \in R} \pi_j = 1.$$

The matrix P_R is $P_R = \{P_{ij} \mid i \in R\}$. We solve this system of equations numerically using an iterative matrix algorithm, the power method [20]. The numerical issues involved in computing this stationary probability distribution, as well as those issues involved in computing the absorption probabilities and mean time to absorption, are discussed in [21].

It is also true that an arbitrary Markov chain with a finite state space has at least one stationary probability distribution over the entire state space. However, if the Markov chain has more than one recurrent class, then any linear combination that sums to one of the stationary probability distributions of the individual recurrent classes is a stationary probability distribution of the chain as a whole.

Our approach is correct regardless of whether the recurrent classes are periodic or aperiodic. Recall that the *period* of a state is the greatest common divisor of all integers $n \ge 1$ for which $P_{ii}^{(n)} > 0$. A state is *aperiodic* if its period is 1, else is *periodic*. All the states in a class have the same period so we can refer to a class as periodic or aperiodic. Only for the states i in an aperiodic recurrent class does the limit $\lim_{n \to \infty} P_{ii}^{(n)}$ exist. However, the long run expected fractions of visits and the stationary probability distribution exist in both cases. Again we assume a finite state space.

D. Resource Usage Estimates

We find, for each recurrent class R, the long run expected fraction of time spent in each state. Then, we find the long run distribution and expectation of each resource with respect to each recurrent class.

Let S be the set of states. Let *RelTime* (S_1) be the long run expected fraction of time spent in state S_1, given that the process is absorbed in class R. From the Markov chain we know the long run expected fraction of visits to each state k, π_k, given absorption in class R. *RelTime* (S_1) can be computed, using the TimeInState function, as follows:

[5]This is not true if the state space is countably infinite. A simple counterexample is a one-dimensional, asymmetric random walk.

$$RelTime(S_1) = \frac{\lim_{n\to\infty} (1/n) \sum_{t=0}^{n-1} E\left[1_{\{X(t)=S_1\}} TimeInState(X(t))\right]}{\lim_{n\to\infty} (1/n) \sum_{t=0}^{n-1} E\left[TimeInState(X(t))\right]} = \frac{TimeInState(S_1)\,\pi_{S_1}}{\sum_{k\in S} TimeInState(k)\,\pi_k}.$$

Recall that 1_A is the indicator random variable for the event A. To show why the last equality holds we derive its denominator. A similar derivation holds for the numerator. Let S be the set of states.

$$\begin{aligned}\lim_{n\to\infty} \frac{1}{n} \sum_{t=0}^{n-1} E\left[TimeInState(X(t))\right] &= \lim_{n\to\infty} \frac{1}{n} \sum_{t=0}^{n-1} \left[\sum_{k\in S} TimeInState(k)\ Pr\left\{X(t)=k\right\}\right] \\ &= \sum_{k\in S} TimeInState(k) \lim_{n\to\infty} \frac{1}{n} \sum_{t=0}^{n-1} Pr\left\{X(t)=k\right\} \\ &= \sum_{k\in S} TimeInState(k)\,\pi_k.\end{aligned}$$

To find the long run distribution of the number of usages of a resource for each absorbing recurrent class, we simply sum over all of the states in the class that use the resource the same number of times, the long run fraction of time spent in that state. To find the long run expected number of usages of each resource for each absorbing recurrent class, we simply take the expectation of the random variable ResUsages:

$$\begin{aligned}E[ResUsages] &= \sum_{k\in S} ResUsages(k)\ Pr\left\{statek\right\} \\ &= \sum_{k\in S} ResUsages(k)\ RelTime(k).\end{aligned}$$

E. Parameters Set of the Stochastic Process

Since firing durations can be zero in the GTPN model, a GTPN can be in two or more states at the same "time." This slightly complicates viewing a GTPN as a stochastic process. In this subsection we discuss that complication and how it can be resolved. The complication is that the right halfline cannot be the parameter set of the GTPN stochastic process. To see this, recall that a stochastic process is a family of random variables indexed by the parameter set and a random variable is a function from the sample space into the reals. If the right halfline were the parameter set, then on some sample path at some parameter t, the random variable $X(t)$ could simultaneously hold more than one value in its range. This contradicts $X(t)$'s being a function.

This complication is resolved by using a different parameter set. The parameter set used is the lexicographically ordered Cartesian product of the nonnegative reals and the natural numbers. The parameters are assigned in the following way. Consider an arbitrary sample path. At any time t in the nonnegative reals, if there are $n (n \geq 0)$ instantaneous state changes, then $X(t, 0)$ is the state before the first (if any) state change, $X(t, 1)$ is the state after the first state change, $\ldots$, $X(t, n-1)$ is the state after the $n-1$th state change, $X(t, m)$, $m \geq n$ is the state after the nth instantaneous state change. Since at most a countably infinite number of instantaneous state changes can occur, the process is never in two or more states at the same 'time." Note that the parameter set of the embedded Markov chain need only be the nonnegative integers with the nth parameter meaning the nth state change.

V. Analysis of the Dining Philosophers

The dining philosopher model is a well-known example which violates net restrictions in previous TPN models. Although performance of this system is largely hypothetical in nature, it serves to illustrate the capabilities of the GTPN analyzer, and it yields some insight into the timing behavior of the dining philosopher protocol.

A GTPN model of the five dining philosophers [13] is shown in Fig. 5. The initial marking of the net shows all five philosophers thinking. We have analyzed the model with deterministic think times, ThinkTime(i), as shown, for each philosopher i. We have also used a slightly modified model (see Fig. 1), to represent think times that are geometrically distributed with mean ThinkTime(i). After thinking, the philosopher competes for two forks which are shared with neighboring philosophers on the left and right, respectively. After acquiring the forks, the philosopher spends a deterministic amount of time eating, DineTime(i). This cycle is repeated as many times as necessary to finish the meal. The firing frequencies f_i associated with transitions that model fork acquisition are used to compute the probabilities that various maximal sets of competing philosophers get the forks they require. These probabilities are calculated as described in Section III-B. Note that all of these transitions are in the same generalized conflict set.

A unique resource is associated with each thinking and dining transition. The GTPN analyzer will compute the long run expected usages of these resources, (ThinkFraction(i) and DineFraction(i)), which correspond to the long run fractions of time that philosopher i spends thinking and dining. From these measures, we can calculate the long run fraction of time philosopher i is idle, waiting for forks (IdleFraction(i) = 1 − ThinkFraction(i) − DineFraction(i)). Performance of the dining philosophers is maximized when time spent waiting for forks and expected time to complete the dinner are minimized.

The dining philosopher model can be analyzed quickly for various think times, dining times, and firing frequencies (i.e., relative aggressiveness in grabbing forks). We first consider the case of two classes of philosophers and deterministic think times. The first class of philosophers, formed by any two nonneighbors, thinks for N units of

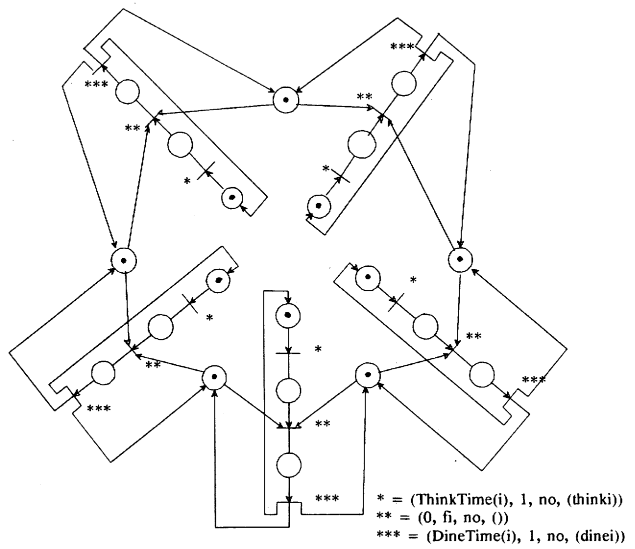
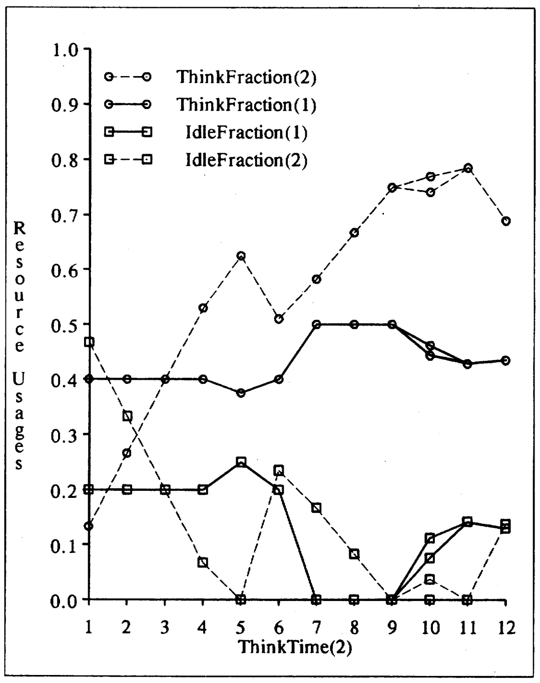

* = (ThinkTime(i), 1, no, (thinki))
** = (0, fi, no, ())
*** = (DineTime(i), 1, no, (dinei))

Fig. 5. Dining philosophers GTPN model.

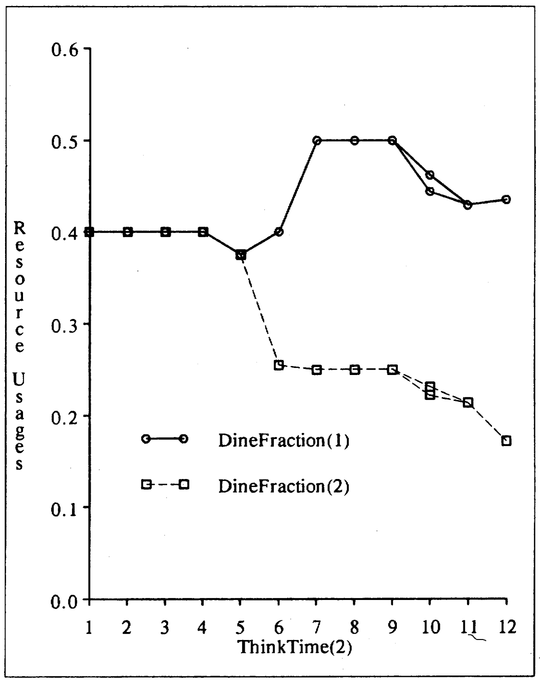
time and dines for N units of time. The second class of philosophers also dines for N units of time. In one experiment, we let $N = 3$, and vary the duration of the think time of the second class of philosophers between 1 and 12 units of time. The firing frequencies, f_i for both classes of philosophers in this experiment are set equal to one. Fig. 6 shows ThinkFraction(i), IdleFraction(i), and DineFraction(i) for the model. Each of the four data points, corresponding to ThinkTime(2) equal to 4, 7, 8, and 9, have two recurrent classes. In each case, both recurrent classes have absorption probabilities equal to 0.5. In each case, the resource usage estimates are the same for both recurrent classes. Each of three data points, corresponding to ThinkTime(2) equal to 5, 10, and 11, have four recurrent classes in the Markov chain. In each case, all four recurrent classes have absorption probabilities equal to 0.25. For ThinkTime(2) equals 5 or 11, the resource usage estimates are also the same for all recurrent classes. For ThinkTime equal to 10, two recurrent classes have identical resource usage estimates which are distinct from the identical resource usage estimates of the other two recurrent classes. Both values are shown in Fig. 6.

Our first observation is that the fractions of time the philosophers spend thinking, waiting, and dining, vary in a complex way with the input parameters in this experiment. Reasoning about the behavior of the system for one parameter setting (i.e., when ThinkTime(2) = 3), shows that after 9 units of time, the system reaches "steady state," in which two philosophers are thinking, two are dining, and one is waiting (interchangeably), forever after. We note that this behavior is highly dependent on the relative delays in the model.

We investigated the complex behavior of system performance as a function of varying think times for the two classes of philosophers further. Assume that each philosopher requires $R = 60$ units of time dining to complete the meal. Let MaxDineFraction be the maximum value of



(a)



(b)

Fig. 6. Performance measures for varying deterministic think times. Two classes of philosophers *ThinkTime*(1) = 3, $\forall j f_j = 1$, *DineTime*(j) = 3.

DineFraction(i) over all i. Then the expected time that the first philosopher(s) complete their meal is $Dmin = R/MaxDineFraction$, which we define to be the "end of the dinner." Fig. 7 shows the total time spent thinking, idle, and dining, and the amount of time needed to complete any unfinished portions of the meal, for a few interesting parameter settings. Starting with a "baseline" model (ThinkTime(i) = DineTime(i) = N), in Fig.

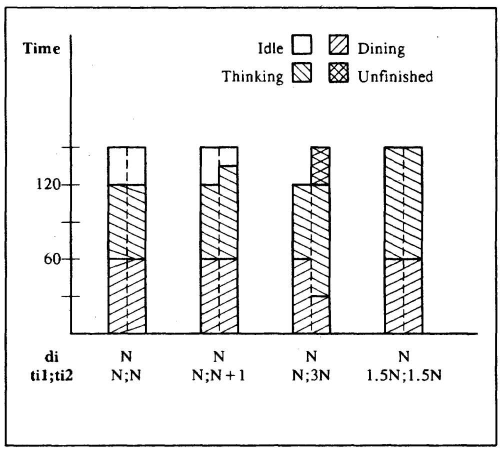


Fig. 7. Performance for selected parameters.

7(a), we see that Dmin = 2.5 hours, of which each class of philosopher spends 60 minutes (40 percent) eating, 60 minutes thinking, and 30 minutes waiting for forks. This corresponds to *ThinkTime*(2) = 3 in Fig. 6. All philosophers finish the meal at the same time. Note that two philosophers in the maximum number that can be dining at the same time, so the baseline model is optimum with respect to *DineFraction*(i) = 0.4. The question is whether the idle time for the philosophers can be reduced while still dining at full capacity. In Fig. 7(b), the second class of philosophers reduce their idle time by slightly increasing their think time by some amount x, $x < N/2$. In Fig. 7(c), the second class of three philosophers increase their think times to $3N$, which reduces idle time to zero for all philosophers, but causes the three to miss half their meal. This corresponds to *ThinkTime*(2) = 9 in Fig. 6. In Fig. 7(d), both classes of philosophers have think times set to *ThinkTime*(i) = 1.5*DineTime*(i) = $1.5N$, which represents the optimum behavior.

For the experiments above, we varied the think time while holding all dining times and firing frequencies constant. In the next experiment, we set the think time of the second class of philosophers to N, and vary the firing frequencies of the first class of philosophers. Fig. 8 gives the results of these analyses. Note that the maximum possible value for DineFraction for the given parameter settings, is 50 percent. When the firing frequency for the aggressive philosophers is 5.0 the fraction of time they spend waiting for forks is reduced by 70 percent (to 0.06).

Finally, we repeated the first experiment (Fig. 6) with geometric think times. Figure 9 shows the performance estimates as a function of mean think time of the class two philosophers. The trends in the performance estimates as *ThinkTime*(2) varies are qualitatively the same as in the deterministic model. However the performance curves are smooth, in contrast to the erratic variations in Figure 6. The erratic performance in the deterministic models is due to cyclic dependencies. These dependencies also make multiple recurrent classes more likely in the deterministic models.

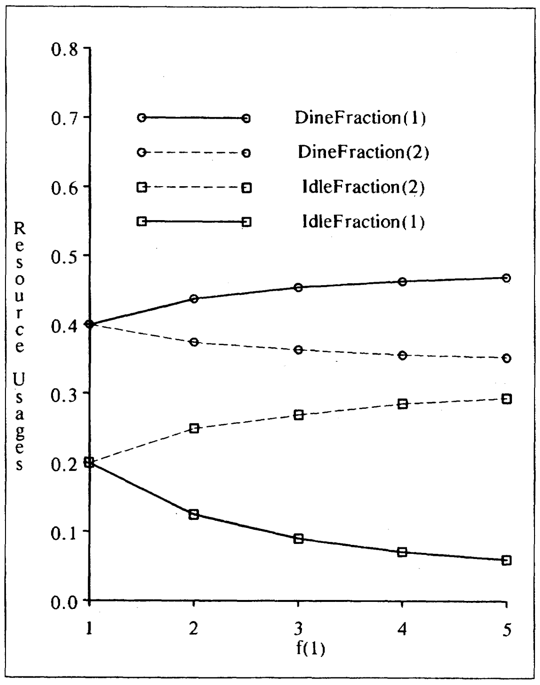


Fig. 8. Performance for aggressive philosophers.

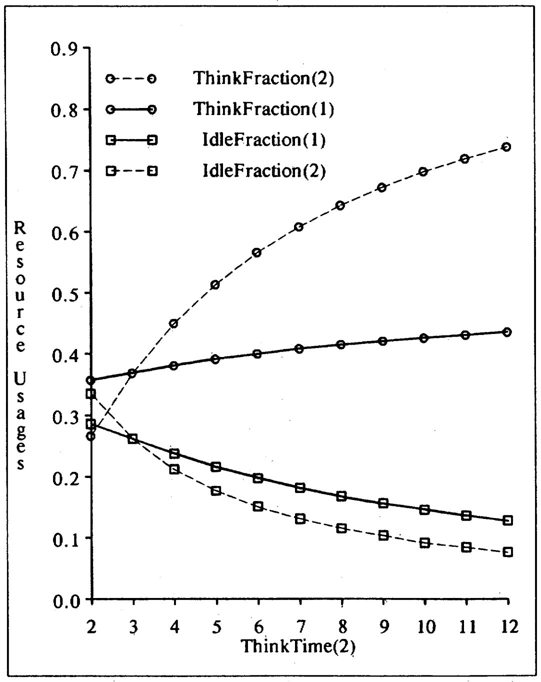
VI. Comparison with SPN Models

Models in which a transition's firing time is an exponential random variable, stochastic Petri Nets (SPN), have been independently proposed by Natkin [5], Symons [6], and Molloy [7], [8]. Marsan, Balbo, and Conte [10] generalized the continuous-time SPN model, GSPN, by allowing transitions which fire in zero time. Molloy [7], [9] also proposed a discrete time SPN model with transition firing times that are geometric random variables. The SPN models are interesting because the reachability graph for these models are (continuous-time or discrete-time) Markov chains. In this section, we compare the conflict resolutions and probability assignment methods of the GTPN model and the SPN models. We then compare the modeling features of the GTPN and SPN models in four respects. Finally, we comment on the complexity issues concerning deterministic firing durations.

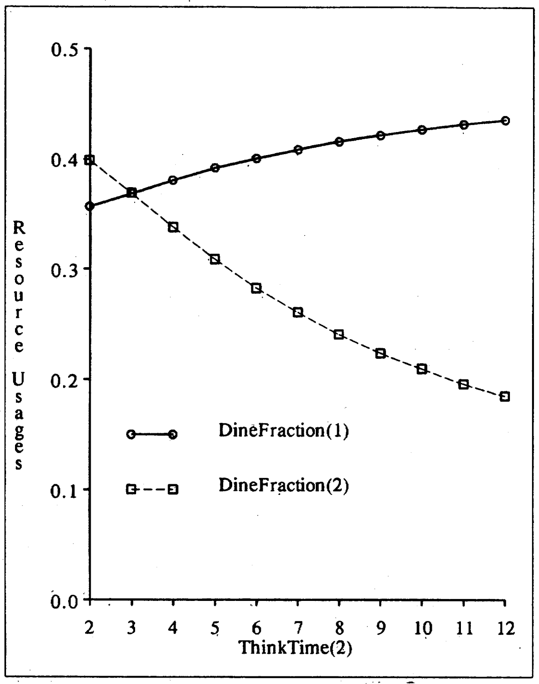
A. Conflict Resolution

The addition of timing information to the Petri net model provides several options for conflict resolution. The method of resolving conflicts in the GTPN is different than the method defined for the GSPN model.

The GTPN conflict resolution submodel uses (possibly state-dependent) *transition firing frequencies* to resolve conflicts. Our underlying assumption is that the conflict is resolved before one of the conflicting transitions starts firing. We also assume that once a transition is in progress, it cannot be preempted by a new conflict. This submodel is useful, for example, if the conflict is due to contention for a shared resource (e.g., two processors requesting use of a shared bus in a multiprocessor).



(a)



(b)

Fig. 9. Varying geometric think times. Two classes of philosophers. *ThinkTime*(1) = 3, $\forall jf_j = 1$, *DineTime*(j) = 3.

In contrast, the conflict resolution submodel for timed transitions in the GSPN is based on *competing transition delays*. The transition which fires first wins the conflict. This mechanism is based on the view that the physical events modeled by the conflicting transition are in progress simultaneously and that the completion of one event disables the other. For example, this view holds if one transition models the successful acknowledgment of a message in a computer network, and the other transition models a timeout process. (Note that the timeout process is exponentially distributed in these models.) The removal

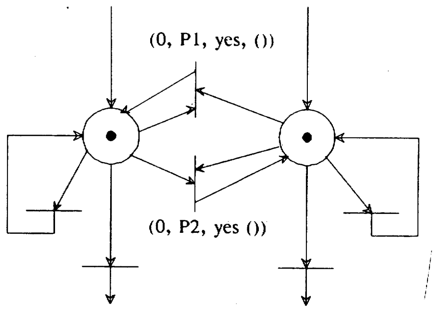


Fig. 10. Competing geometric delays in the GTPN.

of tokens at the time a transition is enabled is not useful for this approach.

Each of the conflict resolution submodels is valid for the corresponding physical systems. Both can be employed in the GSPN and in the GTPN model. For example, the GSPN model uses firing frequencies to resolve conflicts for instantaneous transitions. Conversely, the GTPN can have conflicting geometric holding times such that assigned next-state probabilities are equal to probabilities based on competing delays. To do this, we assign the appropriate competing rate frequencies to instantaneous transitions at the start of a timestep and then model the geometric delays in the usual way. Fig. 10 illustrates this technique. In this figure, $P1 = \lambda_1/(\lambda_1 + \lambda_2)$ and $P2 = \lambda_2/(\lambda_1 + \lambda_2)$ where λ_1 and λ_2 are the means of the competing geometric holding times.

Razouk and Phelps have defined *enabling times* in their TPN. An enabling time specifies a deterministic time that a transition must be enabled before it will start firing. The competing delays in this case are the deterministic enabling time (e.g., a timeout), and a random delay that leads to enabling of a conflicting transition. We plan to extend the GTPN reachability graph construction methods to incorporate enabling times.

Alternative conflict resolution submodels are also discussed in [12] and [22].

B. Probability Assignment

The continuous time GSPN model and the GTPN have an embedded discrete time Markov chain, while the discrete time SPN model itself is a discrete time Markov chain. In either case, probabilities need to be assigned to next states. Each model assigns probabilities that are consistent with its conflict resolution semantics.

When no instantaneous transitions are enabled, the SPN models assign probabilities according to competing transition delays. In the continuous time SPN's, with probability one, no two transitions in progress will finish firing simultaneously. This simplifies probability assignments. In particular, the probability of the next state associated with transition t_i's finishing first has probability equal to t_i's firing rate divided by the sum of the firing rates of the transitions that are in progress.

In the discrete time SPN, with probability greater than zero, two or more transitions in progress can finish firing simultaneously. Though more complicated, it is still possible to assign probabilities to next states using competing transition delays [7], [9]. Unfortunately, using competing delays to assign next state probabilities restricts the allowed probability assignments in the important special case of deterministic firing delays. A deterministic firing delay is represented as a geometric firing delay that has probability one of firing in the next time step. Consequently, using competing delays implies that all the next states have equal probability.

The GTPN uses *transition frequencies* to assign next-state probabilities, independent of transition delays. The transition frequency method, unlike the discrete time SPN, can assign nonuniform probabilities to next states in the case of conflicting transitions with deterministic delays. In general, although the frequency method is powerful, the assignment of static (state-dependent) frequencies which will be used for the dynamic calculation of probabilities, requires careful thought during model construction. The *random switches* used in the GSPN when there are instantaneous transitions firing is a similar method which uses firing probabilities to determine probability assignments. In the random switches method, however, instead of one frequency expression per transition, a probability distribution is explictly given for each possible set of enabled transitions.

C. Modeling Features

The GTPN has capabilities for modeling and analysis lacking in the existing SPN models, in the probability assignments discussed above and in two additional respects: 1) firing durations, and 2) analysis of multiple recurrent classes.

The first respect is firing durations. The GTPN can represent deterministic firing durations which are arbitrary nonnegative real values, including zero. The GTPN can also model geometric holding times as can the discrete time SPN. Since a holding time in the discrete time SPN must be a multiple of some unit step, the GTPN can represent a larger class of firing durations, than the discrete time SPN. Also, since the geometric distribution is the discrete-time analog of the exponential distribution, and since the GSPN cannot represent deterministic delays except with certain strong restrictions [23], the GTPN can also represent a larger class of firing durations than the GSPN model.

The second respect is the GTPN's analysis of multiple recurrent classes. The GTPN allows the performance evaluation of systems that have several possible long run behaviors. In contrast, the GSPN and the discrete time SPN analyses assume that their Markov Chains are irreducible (i.e., have only one recurrent class). We believe that the SPN analysis could be developed to support multiple recurrent classes.

We note that the extended stochastic Petri net (ESPN) model of Dugan, Trivedi, Geist, and Nicola [11] is an SPN model that, in at least one respect, is more powerful than the GTPN. The ESPN allows arbitrary holding times, and has been shown to be useful for analyzing system response to failure. However, the ESPN is analytically tractable only for models with (simple) acyclic reachability graphs, or models where the firing times for all concurrent transitions are exponentially distributed.

D. Complexity Issues

Representing deterministic holding times inherently leads to greater complexity than when holding times are geometrically or exponentially distributed. This is because the memoryless property of the geometric and exponential random variables does not apply. Thus, both the GTPN and the discrete-time SPN model have the potential for large state spaces when deterministic holding times are represented.

The GTPN contains new states for start firing as well as end firing events. If we loosely identify these two state changes as one, there is an equivalence between the states in the GTPN and the discrete-time SPN when geometric holding times are in progress (i.e., one state change per time step, including a cycle back to a given state with the probability that none of the geometric delays completes in the step). When only deterministic holding times are in progress, however, the GTPN may not contain state changes (or new states) for every time step, whereas the discrete-time SPN must. Thus, it appears that the GTPN has at most twice as large a state space as the discrete-time SPN and that for some deterministic models, the GTPN has a smaller state space. We note that the RFT vector, which allows a potential reduction in the size of the state space (in comparison to the discrete-time SPN), adds minimal complexity. It is easy to assign new values to the RFT vector and to find the smallest value in it.

Due to the inherent complexity of deterministic delays, two important goals during our development of the GTPN were to minimize the size of the state space and to minimize the cost of constructing and analyzing it. These goals are reflected in the algorithms and performance analysis techniques presented in Sections III and IV. The GTPN state space is reduced by generating next states for *maximal* set of events that occur simultaneously (including multiple start-firings of a single transition). Another example of reducing the cost of building the state space is our definition of generalized conflict sets (GCS) and the Partition algorithm. Generalize conflicts sets are state independent and thus need only be calculated once. We thus are able to avoid a large fraction of the conflict determination cost when we are calculating the next states of the reachability graph. In contrast, the definition used in the discrete time SPN is based on partitioning enabled transitions into sets that are *in conflict*. This partition is state dependent and thus must be computed for each state.

That we achieved our goal can be seen by example. We have been able to apply the GTPN to obtaining exact performance estimates of multiprocessor memory and bus interference [3]. Previously, researchers have viewed obtaining such exact estimates to be computationally intractable. In contrast, with the GTPN a multiprocessor model with 12 processors, 10 memories, 2 buses, and a geometric time between memory requests with mean of 5 time units has 2026 states and requires 274 seconds to build the reachability graph and analyze it for performance estimates. The largest model we have examined [24] involves a comparison of shared bus cache consistency protocols and has 41 159 states.

VII. Conclusions

We have developed a generalized timed Petri net model for studying the performance of computer systems. The model imposes no restrictions on the net except that the state space be finite. We have also defined methods for performance analysis of the GTPN based on analysis of the embedded discrete-time Markov chain. The analysis is defined for chains which include multiple recurrent classes. Our comparison of this model with the discrete-time SPN and the continuous-time GSPN models shows that the GTPN has capabilities lacking in these other models. In particular, we showed that the GTPN is a better bridge between the TPN and SPN models than is the discrete-time SPN model. We have implemented the GTPN analyzer and demonstrated that the algorithms presented are efficient.

We expect that the capability to obtain accurate analytic performance estimates will yield insight into many computer system models. Such exact estimates have already been obtained with the GTPN for multiprocessor memory and bus interference [3]. Future research plans include studying the applicability of GTPN's to other issues in computer performance evaluation, such as the performance of network protocols, the performance of load balancing algorithms in distributed systems, the performance of database machines, and the performance of advanced architectures for high-speed numeric or symbolic computation.

Acknowledgment

Many people have made helpful comments and suggestions during our research. We would like to especially thank J. Chover, B. Kalsow, T. Kurtz, and U. Manber.

References

[1] W. M. Zuberek, "Time Petri nets and preliminary performance evaluation," in *Proc. 7th Annu. Symp. Computer Architecture*, 1980, pp. 88–96.

[2] R. R. Razouk and C. V. Phelps, "Performance analysis using timed Petri nets," in *Proc. 1984 Int. Conf. Parallel Processing*, Aug. 1984, pp. 126–129.

[3] M. A. Holliday and M. K. Vernon, "Exact performance estimates for multiprocessor memory and bus interference," *IEEE Trans. Comput.*, vol. C-36, pp. 76–85, Jan. 1987.

[4] C. A. Petri, "Kommunikation mit Automaten," Schriften des Rheinisch-Westfalischen Institute fur Instrumentelle Mathematick an der Universitat Bonn, Heft 2, Bonn, W. Germany, 1962; translation: C. F. Greene, Supplement 1 to Tech. Report RADC-TR-65-337, Vol. 1, Rome Air Development Center, Grifiss Air Force Base, NY, 1965.

[5] S. Natkin, "Reseaux de Petri stochastiques," these de Docteur-Ingenieur, CNAM-Paris, June 1980.

[6] F. J. W. Symons, "Introduction to numerical Petri nets, a general graphical model of concurrent processing systems," *Australian Telecommun. Res.*, vol. 14, Jan. 1980.

[7] M. K. Molloy, "On the integration of delay and throughput measures in distributed processing models," Ph.D dissertation, Univ. California, Los Angeles, 1981.

[8] M. K. Molloy, "Performance Analysis using stochastic Petri nets," *IEEE Trans. Comput.*, vol. C-31, pp. 913–917, Sept. 1982.

[9] M. K. Molloy, "Discrete time stochastic petri nets," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 417–423, Apr. 1985.

[10] M. A. Marsan, G. Balbo, and G. Conte, "A class of generalized stochastic Petri nets," *ACM Trans. Comput. Syst.*, vol. 2, pp. 93–122, May 1984.

[11] J. B. Dugan, K. S. Trivedi, R. M. Geist, and V. F. Nicola, "Extended stochastic Petri nets: Applications and analysis," in *Performance 84*, Paris, France, Dec. 1984, pp. 507–519.

[12] M. A. Holliday and M. K. Vernon, "A generalized timed Petri net model for performance analysis," in *Proc. Int. Workshop Timed Petri Nets*, July 1985, pp. 181–190.

[13] J. L. Peterson, *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[14] C. Ramachandani, "Analysis of asynchronous concurrent systems by timed Petri nets," Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, MA, 1974.

[15] C. V. Ramamoorthy and G. S. Ho, "Performance evaluation of asnchronous systems using Petri nets," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 440–449, Sept. 1980.

[16] G. J. Nutt, "Evaluation nets for computer systems perf. analysis," in *1972 Fall Joint Computer Conf., AFIPS Conf. Proc.*, vol. 41. Montvale, NJ: AFIPS Press, 1972, pp. 279–286.

[17] J. G. Kemeny and J. L. Snell, *Finite Markov Chains.* New York: Springer-Verlag, 1976.

[18] R. Sedgewick, *Algorithms.* Reading, MA: Addison-Wesley, 1983, pp. 428–430.

[19] H. M. Taylor and S. Karlin, *An Introduction to Stochastic Modeling.* Orlando, FL: Academic, 1984.

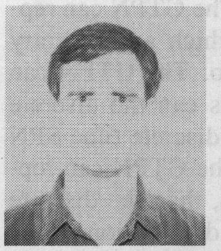
[20] C. H. Sauer and K. M. Chandy, *Computer Systems Performance Modeling.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[21] M. A. Holliday and M. K. Vernon, "The GTPN analyzer: Numerical methods and user interface," Dep. Comput. Sci., Univ. Wisconsin—Madison, Tech. Rep. 639, Apr. 1986.

[22] M. A. Marsan *et al.*, "On Petri nets with stochastic timing," in *Proc. Int. Workshop Time Petri Nets*, July 1985, pp. 80–87.

[23] G. Chiola, "A software package for the analysis of generalized stochastic Petri net models," in *Proc. Int. Workshop Time Petri Nets*, July 1985, pp. 136–143.

[24] M. K. Vernon and M. A. Holliday, "Performance analysis of multiprocessor cache consistency protocols using generalized timed Petri nets," in *Performance 86*, May 1986.



Mark A. Holliday (S'82–M'86) received the B.A. degree with High Honors in mathematics and economics from the University of Virgina, Charlottesville, in 1978, and the M.S. and Ph.D. degrees in computer science from the University of Wisconsin—Madison in 1982 and 1986, respectively.

From 1978 to 1980 he was a programmer at the Johnson Space Center in Houston, TX. From 1980 to 1986 he was a Teaching Assistant and Research Assistant in the Departments of Mathematics and Computer Sciences at the University of Wisconsin—Madison. He is currently an Assistant Professor in the Department of Computer Science at Duke University. His current research interests are in the performance evaluation and design of operating systems for large-scale, shared memory multiprocessors.

Mary K. Vernon (S'82–M'82) received the B.S. degree with Departmental Honors in chemistry in 1975, and the M.S. and Ph.D. degrees in computer science from the University of California at Los Angeles in 1979 and 1982, respectively. As a doctoral student she led the SARA group, under the direction of Prof. Gerald Estrin, from 1980 to 1983.

In 1983 she was a Visiting Assistant Professor in Computer Science at UCLA and a Research Staff member at the Aerospace Corporation in Los Angeles. She is currently an Assistant Professor in Computer Science at the University of Wisconsin—Madison. Her research interests are in performance modeling of distributed and parallel systems, including multicomputers, multiprocessors, and advanced computer architectures for symbolic and numeric computation.

In 1985 Dr. Vernon received the prestigious Presidential Young Investigator Award from the National Science Foundation. She is a member of the Association for Computing Machinery.