Modeling of Concurrent Task Execution in a Distributed System for Real-Time Control

DARTZEN PENG, STUDENT MEMBER, IEEE, AND KANG G. SHIN, SENIOR MEMBER, IEEE

Abstract—**In a distributed system that implements real-time control, computational tasks are distributed over different nodes for execution to improve response time and system reliability. To model system behavior, tasks in each node are first decomposed into *activities*. The activities and precedence constraints among them are then modeled by a generalized stochastic Petri net (GSPN). Finally, a sequence of homogeneous continuous-time Markov chains (CTMC's) is built from the GSPN to model the concurrent task execution in the system.**

The CTMC model is useful for the study of various design and analysis issues in distributed real-time systems. To demonstrate its utility and power, the CTMC model is applied to an important analysis problem: computation of the probability of missing a hard deadline given an activity selection policy and the local state of each node.

Index Terms—**Activity selection, communication primitives, continuous-time Markov chain (CTMC), first passage time, generalized stochastic Petri nets (GSPN), reachability analysis, real-time control, task flow graph (TFG).**

I. Introduction

Due to the availability of inexpensive digital computers with high performance and reliability, most real-time control systems today are realized with digital computers. Such a control system can be divided into two synergistic parts: the *controlled part* or *environment* consisting of a set of control processes,[1] and the *controlling part* consisting of a set of control programs to be executed to direct the behavior of the controlled part [1]. The control programs usually consist of a set of tasks, each of which corresponds to some function to be performed in response to a set of environmental stimuli. The *control mission lifetime* is referred to as the time duration required for the programs, called the *task system*, to start and complete controlling the processes.

Since both response time and system reliability can be improved by using multiple CPU's and memories, distributed systems are attractive candidates to implement real-time control systems. For the generality of our approach, it is assumed that a node in such a distributed system could be a multiprocessor. Many difficult problems need to be solved before the distributed system is realized, such as determining a physical topology of the system, task allocation and scheduling, selecting communication structures, and incorporating reliability considerations.

For applications that do not have strict timing constraints, such as distributed database applications, ad hoc or intuitive solutions are usually acceptable. Any design for time critical processes, however, should be the result of formal analysis and validation to eliminate or minimize the possibility of catastrophic outcomes. Modeling of concurrent execution of control programs, termed *task system modeling*, is an essential step in the formal analysis and design of real-time systems and is the main motivation of this paper.

Conventional task modeling approaches either consider each task as a basic unit or decompose a task into smaller units, i.e., *modules*. The former is called a *task-oriented* model [2], [3], whereas the latter is called a *module-oriented* model [4], [5]. Most of these models, however, have some of the following disadvantages making them insufficient for the formal analysis and design mentioned above:

- The task-oriented model is too coarse. Most tasks communicate with each other during the course of execution, and inter-task communications impose complex precedence constraints which are usually difficult to analyze [2].
- The intertask or internode communication delay is either assumed to be a fixed constant or have a fixed probability distribution. It is impossible to describe the delay under, for example, different communication protocols, or different task scheduling or message handling policies adopted by communication partners.
- It is not easy to describe which task stage each node has been executing. This information is essential when we need to dynamically prioritize the execution of some part of the task system for each node.

In order to remedy the above shortcomings, this paper presents a module-oriented model with a finer granularity[2] than the conventional module-oriented models. Our modeling process consists of the following two steps:

1) Contiguous stretches of executable code of the task system are combined into a set of basic entities called *activities*. A set of activities is formed in such a way that any inherent precedence constraint within the task system and the expected task execution times are preserved.

2) A generalized stochastic Petri net (GSPN) [6] is used to model the activities and their precedence constraints. These

Manuscript received September 1, 1986; revised November 28, 1986. This work was supported in part by the Office of Naval Research under Contract N00014-85-K-0122. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the ONR.

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 8613061.

[1] The term "control process" is used to denote a process (e.g., guidance of a robot arm or an aircraft) to be performed by executing a set of computational tasks.

[2] As one referee pointed out, this will result in the expansion of state space, the price to pay for a finer granularity.

activities are then modeled by a sequence of continuous-time markov chains (CTMC's) [7], [8] by performing *reachability analysis* on the GSPN and assuming independently, exponentially distributed transition firing delays.[3]

The *state* of each CTMC describes the task execution stage each node is in, and a state transition corresponds to the execution of an activity. As will be seen later, the use of a sequence of CTMC's is to facilitate *time-driven* task invocations. The CTMC model offers a useful base on which various problems, such as task response time estimation, optimal message handling policy, and optimal time-out policy, can be rigorously studied.

This paper is organized as follows. Section II describes the task system and states some simplifying assumptions. Section III introduces necessary concepts, definitions, and notation. The problem statement and the proposed modeling process are presented in Section IV. The probability of missing a hard deadline is computed in Section V to demonstrate the use of our model. Finally, the paper concludes with a few remarks in Section VI.

II. The Task System

Tasks in a distributed real-time system can be classified into two types: *periodic* and *nonperiodic*. A periodic task is invoked at fixed time intervals and constitutes the normal computation for the processes under control. A nonperiodic task can be invoked at any time, especially for abnormal or critical situations. In this paper, we consider only periodic tasks since they are not only the nucleus but also the unique feature of real-time tasks.

Most periodic tasks cooperate with each other through communication to accomplish the overall control mission. The communication between two cooperating tasks is usually related to precedence constraints between them. These constraints are in the form that the completion of some parts of one task enables some parts of the other task to be ready for execution. To identify a set of tasks with precedence constraints, all tasks in the system can be partitioned into mutually exclusive sets of tasks called *precedence classes*. Two tasks are in the same class if and only if they communicate, directly or indirectly, with each other.

For the purpose of modeling, we proceed with the following assumptions:

A1) At any given time, all resources in the system are dedicated to a single control mission.

A2) Tasks are preassigned to the nodes, and remain unchanged throughout the control mission lifetime.

A3) Each nonperiodic task alone forms a single precedence class.

A1) indicates that no other control mission can begin execution before the current one is completed. This assumption excludes a rare and more complex case where more than one control mission simultaneously compete for system resources. A2) is usually the case in practice due to the time overhead associated with on-line assignment of tasks. A3) is to simplify the treatment of the task system. Specifically, if each nonperiodic task alone forms a distinct class, by definition, it does not communicate with any other task and, thus, imposes no precedence constraints in the task system. Notice that the above assumptions do not exclude the case where the periodic tasks may form more than one precedence class.

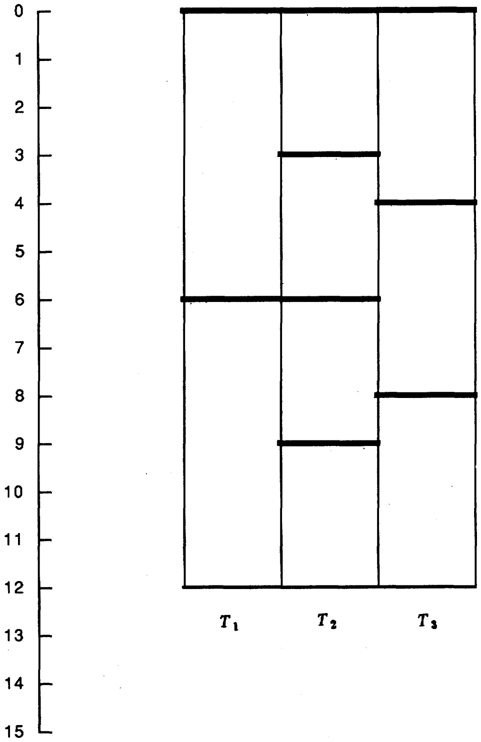


Fig. 1. A planning cycle of a task system with three tasks.

To analyze normal system behavior, a cycle of the task system, called the *planning cycle*, is identified such that the task system can be fully characterized in one planning cycle. The least common multiple (LCM), L of $\{d_i | i = 1, 2, \cdots v\}$ can be used for calculating the planning cycle, where $1/d_i$ is the frequency of triggering a periodic task T_i and v the total number of periodic tasks in the task system. The planning cycle is then defined as the time interval $[t_0 + kL, t_0 + (k + 1)L)$ where t_0 is the time the mission starts and k is a nonnegative integer. Notice that each task T_i is invoked $r_i = L/d_i$ times in a planning cycle, and there may be a (possibly different) hard deadline associated with each invocation of a task. The example in Fig. 1 shows a system with 3 tasks T_1, T_2, and T_3, with $d_1 = 6$, $d_2 = 3$, and $d_3 = 4$ time units, all of which are first invoked at $t_0 = 0$. A planning cycle is the time interval [0, 12) in which T_1, T_2, and T_3 are invoked 2, 4, and 3 times, respectively.

It is also assumed that no pipelining of tasks is allowed. The current invocation of a task must be completed before its next invocation; if a task is not completed prior to its next invocation, it is simply discarded. Since the process under control may have changed, for example, its sensor values by the time of the next invocation of a task, there is no need to execute a previous invocation of the task with obsolete sensor data.

III. Definitions and Notation

This section introduces concepts, definitions, and notation that will be used throughout this paper. The *task flow graph*

[3] The set of transitions that can be fired is time dependent, however.

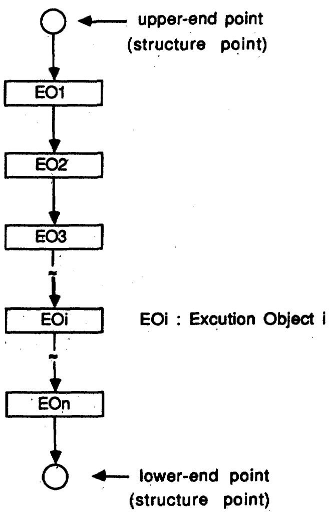


Fig. 2. A chain.

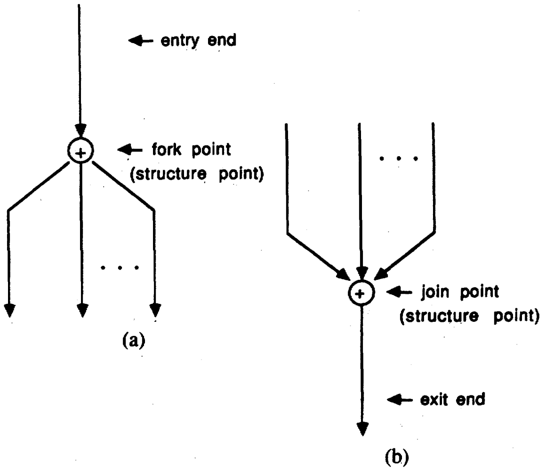


Fig. 4. An Or-Subgraph. (a) Or-Fork. (b) Or-Join.

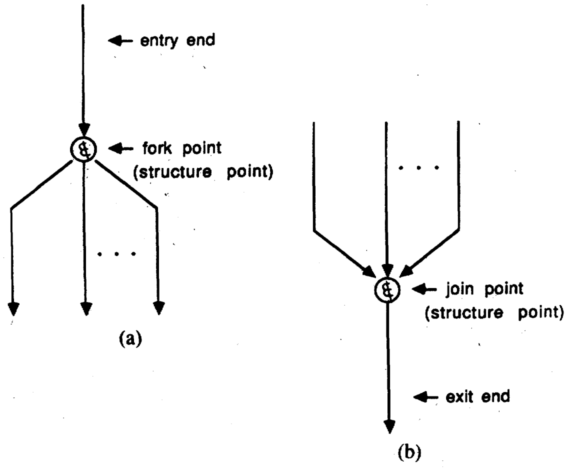


Fig. 3. An And-Subgraph. (a) And-Fork. (b) And-Join.

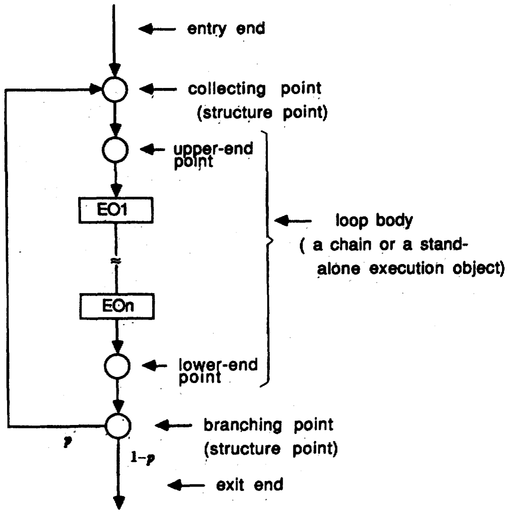


Fig. 5. A loop.

(TFG), and its *task tree*, which serve as the input to our modeling process, are first presented and then followed by a discussion of the combination and expansion phases which deal with modules.

A TFG describes a task to be executed by a node in the distributed system. The TFG is composed of four types of subgraphs: *chain*, *And-Fork* to *And-Join*, *Or-Fork* to *Or-Join*, and *loop*.

A chain (Fig. 2) is the largest possible concatenation of multiple entities called *execution objects*. A *communication point* is an execution object which represents one of the six communication primitives (to be elaborated on in Section IV) used in the system or an output signal sent to the processes under control. A sequential code stretch or a communication point is called a *basic* execution object, and a single execution object, which is not a chain by definition, is called a *stand-alone* execution object. A stand-alone execution object may be a basic execution object, an And-Fork to And-Join subgraph, an Or-Fork to Or-Join subgraph, or a loop.

An And-Fork to And-Join subgraph (or simply called an And-Subgraph) (Fig. 3) consists of more than one branch, all of which must be executed (possibly in parallel). A branch of the And-Subgraph may be a stand-alone execution object or a chain.

Similarly, an Or-Fork to Or-Join subgraph (or simply called an Or-Subgraph) (Fig. 4) consists of more than one branch. However, one and only one branch of the Or-Subgraph is executed, and the probability of choosing each branch is assumed to be given. Another point that differentiates an Or-Subgraph from an And-Subgraph is that a branch of the Or-Subgraph could contain no execution object at all.

A loop (Fig. 5) consists of a loop body with a "looping back" probability p. Like a branch of the And-Subgraph, the loop body may be a stand-alone execution object or a chain.

Also shown in Figs. 2–5 are the upper end and the lower end points of a chain, the fork and join points of an And-Subgraph and an Or-Subgraph, and the collecting and branching points of a loop. Since these points serve as boundaries of their respective subgraphs, they are called *structure points*. Note that there is no structure point defined for a basic execution object, and that there may or may not be stucture points for a stand-alone execution object, depending on the specific execution object it represents.

A task tree describes how the TFG is organized by the four

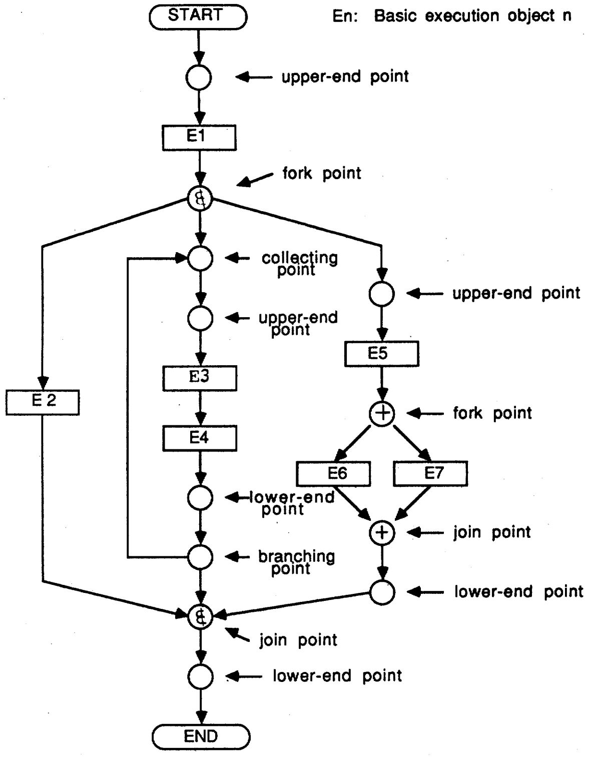


Fig. 6. A task flow graph (TFG).

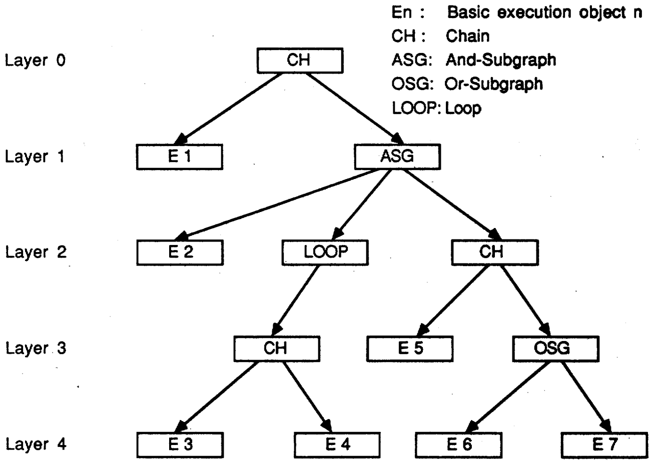


Fig. 7. The task tree for the TFG in Fig. 6.

types of subgraphs and the basic execution objects mentioned above. The root of the task tree is the TFG itself, the leaves are the basic execution objects while the nonleaf nodes are the four types of subgraph. A *layer number* is a nonnegative integer assigned to each node in the task tree to indicate the relative position of the node in the tree and, consequently, of the subgraph and the basic execution object in the TFG. A higher layer number is assigned to a node of an inner layer, while a lower number is assigned to a node of an outer layer, and the lowest layer number 0 is assigned to the node of the outermost layer, the TFG itself.

For example, Fig. 6 is a TFG whose task tree is shown in Fig. 7. The TFG in Fig. 6 as a whole is a chain which consists of two execution objects: $E1$ and an And-Subgraph. The And-Subgraph has three branches: the first is a stand-alone (and basic) execution object $E2$, the second is a stand-alone execution object (a loop), while the third is a chain which contains two execution objects: $E5$ and an Or-Subgraph. The loop body in the second branch is also a chain consisting of two basic execution objects $E3$ and $E4$. Since $E1$ and the And-Subgraph are one layer "inside" the chain (or the chain is said to be one layer "outside" $E1$ and the And-Subgraph) representing the TFG itself, each of them is assigned the layer number 1. Likewise, each of the three branches of the And-Subgraph is one layer inside the And-Subgraph, so each of the branches is assigned layer number 2, and so on. Note that an outer layer node contains all inner layer nodes under itself. For example, the chain on the third branch of the And-Subgraph contains $E5$, the Or-Subgraph and, thus, $E6$ and $E7$.

In a chain, an execution object is not ready for execution until the one immediately preceding it has been completed. In an And-Subgraph, however, no precedence constraints are needed among its branches. Because of the constructs of an Or-Subgraph and loop, it is not possible to use a relation such as *partial ordering* to describe the precedence constraints between any two basic execution objects.

A *module* is an entity resulted from combining a set of two or more code stretches or modules.[4] A combination is said to *retain the precedence constraints* among the original stretches if and only if the combined set belongs to one of the two types: 1) a set of contiguous modules on a chain, and 2) a set of branches of an And-Subgraph or Or-Subgraph. All combinations to be discussed here are of this type. When a set of modules is combined into a module e, e is said to be an *equivalent* module of the combined set. The largest module that can be formed without violating any precedence constraint is called an *activity*. Depending on how far the combination process can go in the TFG, the boundary of an activity may or may not be a communication point. The boundary of an activity which is not a communication point is called a *control point*. After all the activities are found, the resulting graph is called a *communication flow graph* (CFG). More on this will be discussed in Section IV.

We have introduced the TFG, its task tree and the combination process which are necessary tools for our modeling process. In what follows, a brief discussion on a GSPN [6] and related definitions necessary to our modeling process is given.

A Petri net (PN) [9] is a four-tuple, $C = (P, T, I, O)$ where $P = \{p_1, p_2, \cdots, p_n\}$ is the set of places, $T = \{t_1, t_2, \cdots, t_m\}$ the set of transitions, $I{:}T \rightarrow P$ the input function, and $O{:}T \rightarrow P$ the output function. A marking $\mu{:}P \rightarrow \mathbf{I}^+$ represents the number of tokens for each place $p_i \in P$ where $\mathbf{I}^+$ is the set of nonnegative integers. $M = (P, T, I, O, \mu)$ is a PN with a marking μ. After being enabled, a transition fires by removing one token from each of its input places and adding one token to each of its output places.

A GSPN is a type of marked PN where a nonnegative random variable representing the firing delay is associated with each transition. Depending on its firing delay, each transition of the GSPN can be classified into one of the two

[4] This is a recursive definition. Further, we will use the term "module" to refer to both the code stretch and module in the rest of the paper.

types: *immediate* and *timed*. A transition is immediate if its firing delay is zero with probability one, and timed otherwise. A place p_i in the GSPN is said to be *instantaneous* if a transition with p_i as a sole input place is immediate, and *noninstantaneous* otherwise. The state of the GSPN is the marking of the set of all noninstantaneous places. A state is *vanishing* if it enables at least one immediate transition and *tangible*, otherwise.

The procedure to find the reachability set S of states from some initial marking μ is called the *reachability analysis* of the GSPN. A noninstantaneous place p_i of a GSPN with an initial marking μ is *safe* if $\forall \mu' \in S$, $\mu'(p_i) \leq 1$. A GSPN is safe if every noninstantaneous place in the net is safe. As will become clearer, all GSPN's considered in this paper are safe. The remaining part of this section will deal with the CTMC model describing the task system.

Let M be the set of all nodes in the system, each of which is assigned *a priori* a set of tasks to be executed, and $Z(m)$ be the number of processors of node $m \in M$. Assuming that, in a planning cycle of the task system, tasks are invoked at times $w_1, w_2, \cdots, w_l$ where $w_1 = t_0$, the beginning of the cycle, and $0 \leq w_1 < w_2 < \cdots < w_l < w_{l+1} = t_0 + L$, a sequence of CTMC's, $\{(S_k, \Lambda_k, \Theta_k) | k = 1, 2, \cdots, l\}$, is necessary to model the task system.

State space S_k is the set of states of the CTMC model reachable during time interval $I_k = [w_k, w_{k+1})$. Then, $S = \cup_{k=1}^{l} S_k$ is the *total* state space.

$\Lambda_k: S_k \times S_k \rightarrow T$ is the event-driven transition function between two states in S_k. Each element in T is a triplet (λ, ξ, m) where $\lambda \geq 0$ is the transition rate whose inverse represents the expected activity execution time, ξ the prespecified branching probability, and m the node to execute the activity. Let λ_{ij}, ξ_{ij}, and m_{ij} denote the transition rate, the branching probability, and the node associated with states s_i and s_j, respectively. A transition with $\lambda_{ij} = 0$ is trivial, meaning no transition between these two states. A set of nontrivial transitions from s_i to some other states s_j such that $\Sigma_j \xi_{ij} = 1$ is called a *branching set*, and each transition in the set with $\xi_{ij} < 1$ is called a *branching* transition. On the other hand, a transition with $\xi_{ij} = 1$ is called a *lone* transition. As will be seen in Section IV, a branching set results from either an Or-Fork or a loop.

A set of event-driven transitions associated with $s_i \in S_k$ and node $m \in M$ is defined as: $\text{OUT}(s_i, m) = \{(\lambda_{ij}, \xi_{ij}, m) | s_j \in S_k, \lambda_{ij} \neq 0\}$, the set of all nontrivial transitions from s_i to be fired in node m during I_k. Given that the task system is in s_i, the number of transitions, $|\text{OUT}(s_i, m)|$,[5] may be larger than the number of processors, $Z(m)$, for some node $m \in M$. Therefore, a decision must be made as to which activities/transitions will be chosen[6] to fire. An *activity selection policy* δ specifies this choice for each $m \in M$ and for each $s_i \in S$. Let D_i denote the set of all decisions available at s_i. The *policy space* Δ is defined as the set of all such policies, i.e., $\Delta = D_1 \times \cdots \times D_i \times \cdots \times D_n$ where $n = |S|$.

Given the policy δ, the set of transitions chosen for concurrent firing at s_i is called an *active transition set*, ATS_i^δ. Note that ATS_i^δ is a subset of $\text{OUT}(s_i, *) = \cup_{m \in M} \text{OUT}(s_i, m)$.

$\Theta_k: S_k \rightarrow S_{k+1}$ is the *time-driven* transition function. All transitions specified by Θ_k occur at time w_{k+1} and take no time to complete the transitions. Specifically, Θ_k specifies which state in S_{k+1} to start with at time w_{k+1} for each state in S_k upon some task invocations.

IV. The Modeling Process

Given the task flow graphs (TFG's) and their task trees for each node, our objective is to model the task system with a sequence of CTMC's such that:

C1) the resulting CTMC model has as small a state space as possible,

C2) the precedence constraint between any two basic execution objects in the original TFG is retained, and

C3) the expected execution times for each task invocation and for the task system as a whole are preserved in the resulting CTMC model.

C1) is concerned with optimality, while C2) and C3) are constraints. Since the size of the total state space S is generally an exponential function of the number of activities, modules should be combined, whenever possible, to build the smallest set of activities while satisfying C2) and C3). This combination may cause the following two problems: P1) loss of potential parallelism, and P2) dependence among resulting activities. P1) is due to the combination of modules in an And-Subgraph. This is acceptable since analysis based on such a combination will err on a conservative side in meeting hard deadlines. P2) is caused by the combination between modules inside and outside an Or-Subgraph. A combination of this kind merges an outside module with a module on each branch of the Or-Subgraph, and, thus, introduces dependence between the resulting modules on any two branches. This calls for the use of a stochastic process with mutually dependent transition delays. However, since such a combination could drastically reduce the size of state space, we choose to approximate the model by ignoring this effect.

The overall procedure to build the proposed CTMC model of the task system is divided into three sequential stages: building the smallest activity set of a TFG, constructing a GSPN of concurrent task execution, and deriving the CTMC model from the GSPN.

A. The Smallest Activity Set

Given the TFG's and their task trees for each node, the set of activities of each TFG is determined by identifying their boundaries. We propose a procedure consisting of N iterations of *combination* and *expansion* phases where N is the largest layer number in the task tree of interest. The combination phase merges modules within subgraphs of the same layer while the expansion phase penetrates the boundaries of the subgraphs being worked on and performs some preprocessing for the next combination phase.

Starting from the second innermost (i.e., layer $N - 1$), and

[5] Each branching set as a whole is counted as a single transition.

[6] If $Z(m) \geq |\text{OUT}(s_i, m)|$, then all elements in $\text{OUT}(s_i, m)$ are chosen.

[7] As mentioned before, potential parallelism is lost if, for example, there are more than two processors available to execute in parallel any two branches to be combined.

working through the outer layers, each iteration works on all subgraphs in the layer until the outermost layer is expanded. The layer which is currently being worked on is called the *active layer*.

1) The Combination Phase: To meet C2), only three types of combination are allowed: *vertical*, *horizontal*, and *total* combinations. Vertical combination merges vertically adjacent modules of a chain into a new equivalent module. Horizontal combination operates on two or more branches in an And-Subgraph or an Or-Subgraph. Total combination merges the whole subgraph into a single equivalent module as was done in [5].

A chain can have vertical and total combinations, an And-Subgraph or an Or-Subgraph can have horizontal and total combinations, while a loop can have total combination only. These combinations are performed only in the subgraphs in the active layer of the TFG.

To satisfy C3), the expected execution time of an equivalent module is computed as follows. The expected execution time of the equivalent module after a vertical combination in a chain or the horizontal combination in an And-Subgraph[7] is the sum of those of all the component modules combined into it. For the horizontal combination in an Or-Subgraph, the branching probability p_e and the expected execution time $E[T_e]$ of the equivalent module e are determined by the following two equations.

$$p_e = \sum_{i \in G} p_i \tag{1}$$

$$E[T_e] = \frac{\sum_{i \in G} E[T_i] p_i}{p_e}, \qquad p_e \neq 0 \tag{2}$$

where G is the nonempty set of modules combined into the equivalent module e, p_i, and $E[T_i]$ are, respectively, the branching probability and the expected execution time of module i in G. Note that both (1) and (2) also hold for the total combination of an Or-Subgraph. For a loop, however, the expected execution time $E[T_e]$ after the total combination is

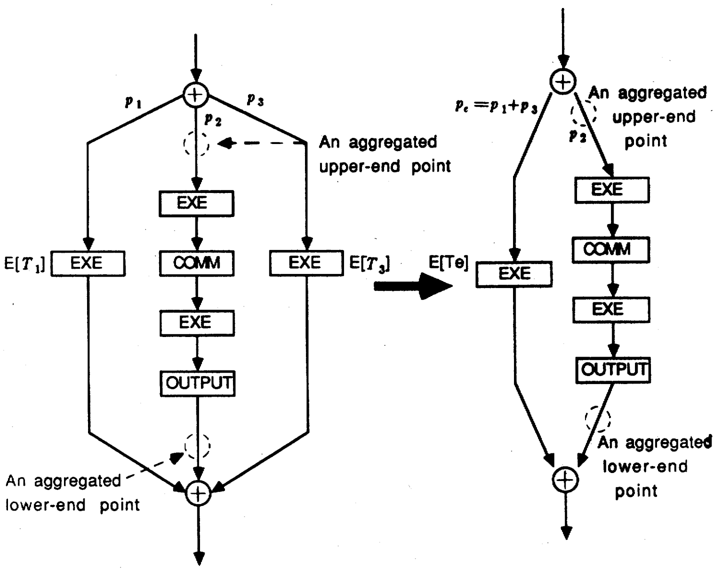
$$E[T_e] = \frac{E[T_l]}{1-p}, \qquad p \neq 1 \tag{3}$$

where $E[T_l]$ is the expected execution time of the single equivalent module in the loop body.

To facilitate the next phase, if a total combination is performed in any of the four types of subgraph, the subgraph is replaced with its equivalent module, and its two structure points are removed.

For example, Fig. 8 illustrates the horizontal combination of an Or-Subgraph, and Fig. 9 shows the total combination of a loop. In the figures, $\boxed{\text{COMM}}$ represents a communication primitive, $\boxed{\text{OUTPUT}}$ is the point where an output signal is sent, and $\boxed{\text{EXE}}$ represents a module. Also indicated in Fig. 8 are structure points which have been aggregated in the previous expansion phase (see below).

2) The Expansion Phase: The expansion phase is to remove the structure points of the subgraphs of the active layer such that the next combination phase can be applied across the original boundaries. This is done by applying two operations: *module migration* and *structure points aggregation*. Modules outside an Or-Subgraph and adjacent to its join point are moved inside the subgraph only if the next combination phase can reduce further the total number of resulting activities. This is the case when there is a module on each branch of the Or-Subgraph ready to be combined with the migrated module. Two adjacent structure points are aggregated into one if the aggregation should decrease the number of instantaneous places in the resulting GSPN model.


p_1
p_2
p_3
An aggregated upper-end point
EXE
$E[T_1]$ EXE
COMM
EXE $E[T_3]$
EXE
OUTPUT
An aggregated lower-end point
$p_e = p_1 + p_3$
An aggregated upper-end point
p_2
EXE
E[Te] EXE
COMM
EXE
OUTPUT
An aggregated lower-end point

$$E[T_e] = \frac{E[T_1]p_1 + E[T_3]p_3}{p_e}$$

Fig. 8. The combination process for an Or-Subgraph.

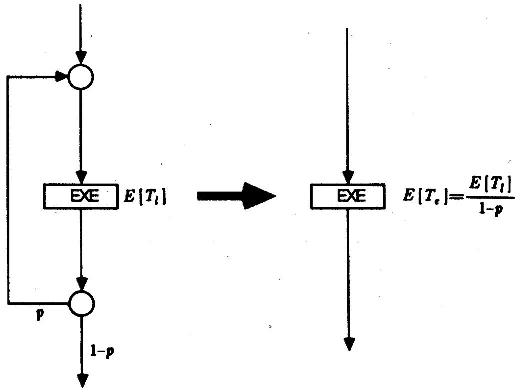


Fig. 9. The total combination of a loop.

While performing the above two operations, control points need to be assigned and communication points to be identified to "stop" further combination on modules located between these points in order not to violate the precedence constraints. Therefore, each of these "stopping points" will serve as the boundary of the two activities, if any, on each side of the point, and no other point in the TFG will serve the same purpose.

The rules for module migration are summarized as follows.

R1) Vertically combine all contiguous modules in the active layer into their equivalent modules as in the combination phase described above.

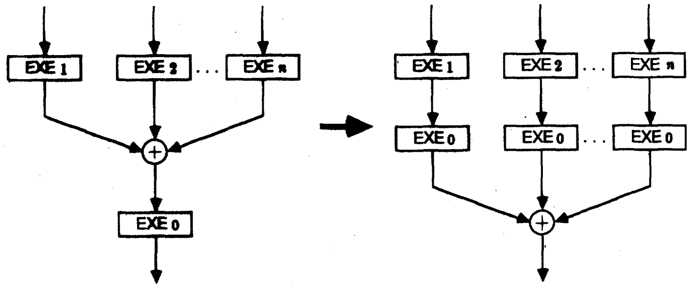


Fig. 10. The movement of a module into an Or-Subgraph.

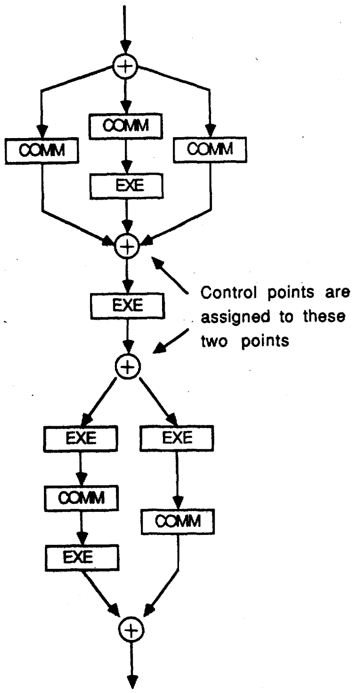


Fig. 11. A module between two Or-Subgraphs.

R2) If a module is outside only one Or-Subgraph whose join point is adjacent to the module, move the module into the subgraph if there is a module adjacent to the join point on each branch of the Or-Subgraph[8] (Fig. 10).

R3) If a module is outside only one Or-Subgraph whose fork point is adjacent to the module, or the module is between two Or-Subgraphs, assign control point(s) to the structure point(s) associated with the subgraph(s) without moving the module (Fig. 11).

R4) For any module that is adjacent to an And-Subgraph or a loop, assign a control point to the adjacent structure point of the subgraph without moving the module (Fig. 12).

R5) For any subgraph that is adjacent to a communication point, assign a control point to the adjacent structure point of the subgraph without moving the module.

After module migration is performed, structure points should be aggregated as required to eliminate redundant structure points and prepare for the next combination phase. Apparently, different modeling bases have different modeling efficiency, making the conditions for aggregation different.

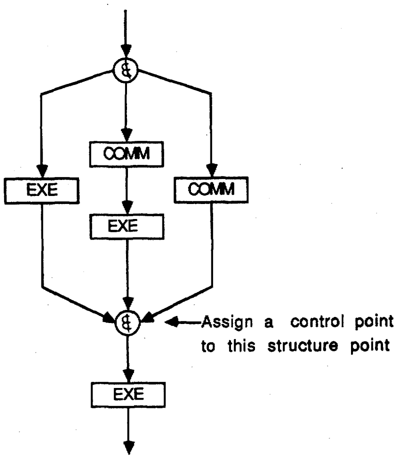


Fig. 12. A module adjacent to an And-Subgraph.

For example, a structure point regarded redundant by a more efficient modeling base, such as stochastic activity networks [10], may not be regarded redundant by a less efficient base such as a GSPN,[9] whose modeling efficiency is limited by its execution rules. Since the GSPN cannot efficiently model a logic structure which is more complex than the fork or join point of an And-Subgraph or Or-Subgraph, a structure point is regarded redundant only if aggregation on this point should result in a logic structure which is logically less complex than AND Fork, AND Join, OR Fork, and OR Join. (Each of these is called an "AND/OR logic").

The notation $P1 \rightarrow P2$ is used to denote a case of two adjacent structure points, where $P1$ is immediately followed by $P2$. Depending on the relative positions of $P1$ and $P2$ in the TFG, three different classes can be identified:

- $P2$ is in the active layer, which is one layer inside the layer of $P1$.
- $P1$ is in the active layer, which is one layer inside the layer of $P2$.
- Both $P1$ and $P2$ are in the active layer.

One example of structure point aggregation of each of the above classes is shown in Fig. 13 where $A1$, $A2$, $R1$, and $R2$ represent the fork and join points of the And-Subgraph and the Or-Subgraph, respectively. When the inner Or-Subgraph is expanded (Fig. 13(a)), the inner $R1$ is redundant and, thus, aggregated into the outer $R1$ to form an AND/OR logic. In order to include new branching probabilities after aggregation, each branching probability in the inner Or-Subgraph should be adjusted by multiplying each branching probability in the inner subgraph by their probability in the outer subgraph. In this case, since further combination may still be possible, no control point is assigned. In Fig. 13(b), the inner And-Join cannot be aggregated into the outer Or-Join to form an AND/OR logic, implying that no aggregation is possible and, thus, a control point is assigned to the inner And-Join to represent a boundary between certain activities. Fig. 13(c) shows an Or-Subgraph immediately followed by an And-Subgraph in the

[8] This is the case that introduces a dependence relationship among combined modules on any two branches of the Or-Subgraph.

[9] Irrespective of the various modeling bases used, the resulting CTMC models are the same. However, using a more efficient modeling base makes the corresponding reachability analysis more difficult.

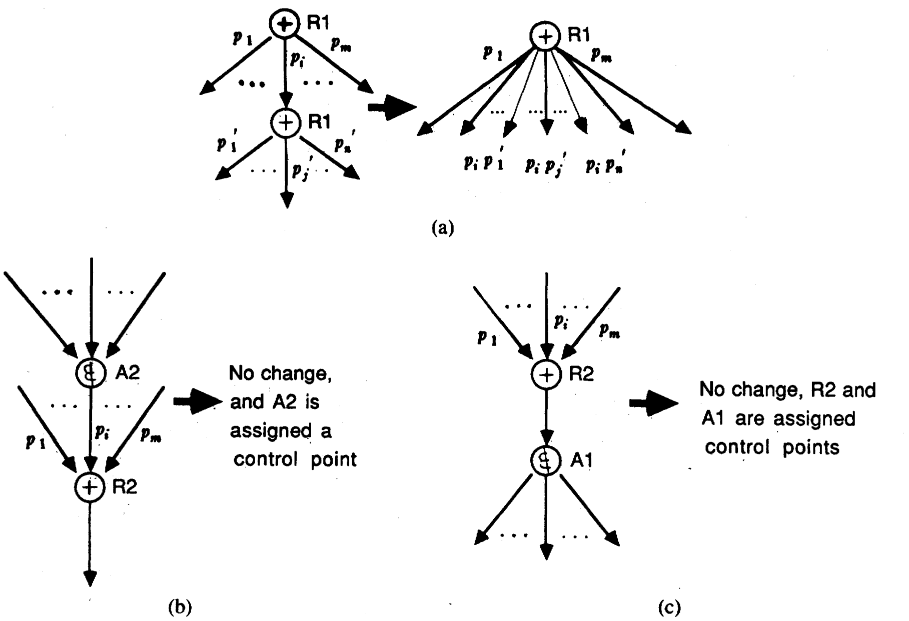


Fig. 13. Three examples of aggregation rules. (a) Case (12). (b) Case (23). (c) Case (34).

same layer. Since no AND/OR logic can be formed by aggregating these two structure points, they remain unchanged, except that each of them is assigned a control point.

Since only four types of subgraph are considered, it is not difficult to i) enumerate all different cases in which two structure points are adjacent to each other, and ii) set up the aggregation rules for each case. All 39 cases and their aggregation rules are given in the Appendix.

Upon completion of the current iteration of both phases, the active layer is raised outward by one so that the next iteration can be applied on the new active layer. This procedure is repeated until the outermost layer is finally expanded. Fig. 14 is the summary of the process described thus far. Clearly, since the precedence constraints and the expected execution times are preserved under each operation, $C2$ and $C3$ are both satisfied, and the resulting set of modules is the minimal set of activities. Because the decomposition process is *communication-oriented*, the resulting graph is called a *communication flow graph* (CFG).

As an example, the resulting CFG for the TFG in Fig. 15 is shown in Fig. 16. Notice that the CFG does not maintain the structure of the original TFG. The CFG is just a collection of activities, communication and control points properly organized by AND/OR, sequential and loop logic structures and, more importantly, no longer contains the four types of standard subgraph.

B. GSPN Representation

To fully describe the task system with a GSPN, the following three aspects of the task system must be properly modeled and incorporated.

F1) The precedence constraints among activities within a single task, which are imposed by the corresponding CFG.

F2) The precedence constraints among activities of two communicating tasks, which are imposed by the semantics of the communication primitives used.

```
PROCEDURE combination_phase( layer_num)
BEGIN /*** combination phase for the current active layer ***/
FOR each subgraph whose layer number = layer_num DO
    BEGIN / ***combination phase for one subgraph ***/
        Perform vertical, horizontal, and total combinations on modules in the
        subgraph as described in Section IV-A-1).
    END /*** combination phase for one subgraph ***/
END /*** combination phase for the current active layer ***/


PROCEDURE expansion_phase( layer_num)
BEGIN /*** expansion phase for the current active layer ***/
    Identify all communication points whose layer number = layer_num.
    FOR each subgraph whose layer number = layer_num DO
        BEGIN /*** code stretch movement for one subgraph ***/
            Move modules whose layer number = layer_num into the subgraph
            according to the rules R1, R2, R3, R4 and R5 as described in Section IV-A-2).
        END /*** code stretch movement for one subgraph ***/

    FOR each structure point whose layer number = layer_num, and has no module
        adjacent to it DO
        BEGIN /*** structure point aggregation ***/
            Aggregate the structure point, and assign control point according the rules
            listed in the Appendix.
        END /*** structure point aggregation ***/
END./*** expansion phase for current active layer ***/


/******************* Main Program ***********************/
BEGIN /*** Decomposition Process ***/
    FOR active_layer = N-1 to 0 DO
        BEGIN /*** 1 iteration of combination and expansion phases ***/
            combination_phase( active_layer );
            expansion_phase( active_layer );
        END /*** 1 iteration ***/
END /*** decomposition process ***/
```

Fig. 14. The summary of the process to build the smallest activity set.

F3) The time-driven, rather than event-driven, task invocations.

F1) and F2) determine the structure, whereas F3) affects the initial markings, of the resulting GSPN. F1) requires to model sequential, AND/OR, and looping logics on the activities in a CFG. This modeling process is straightforward and, thus, omitted here (interested readers may consult [9] and [11]). F2) and F3) are addressed in Sections IV-B-1) and IV-B-2), respectively.

1) The GSPN Models of Communication Primitives: The communication primitives to be modeled include SEND-RECEIVE-REPLY, QUERY-RESPONSE, and WAITFOR. Although these primitives were proposed in [12] for intertask communications in an integrated multirobot system (IMRS),

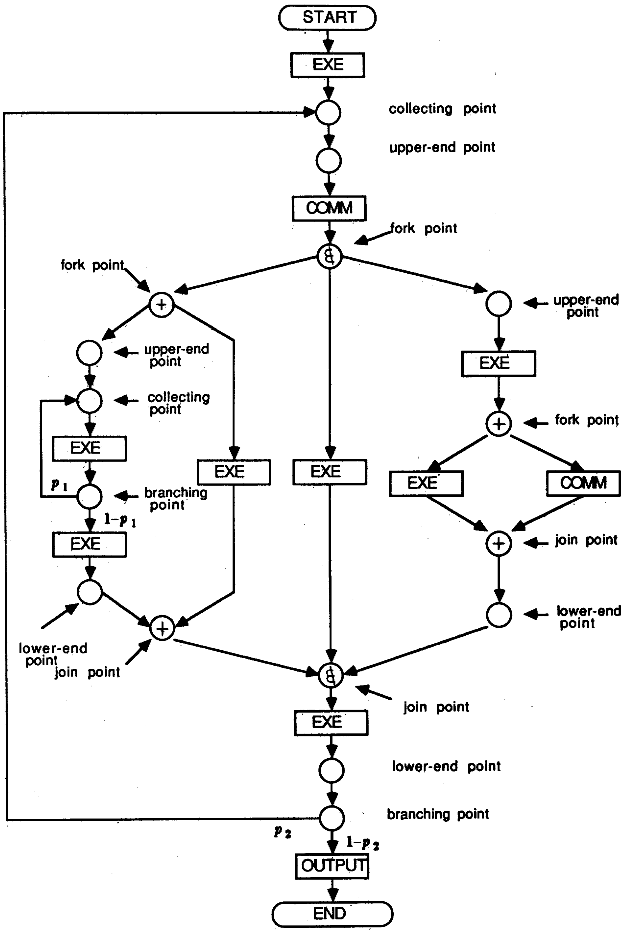


Fig. 15. The TFG of a node.

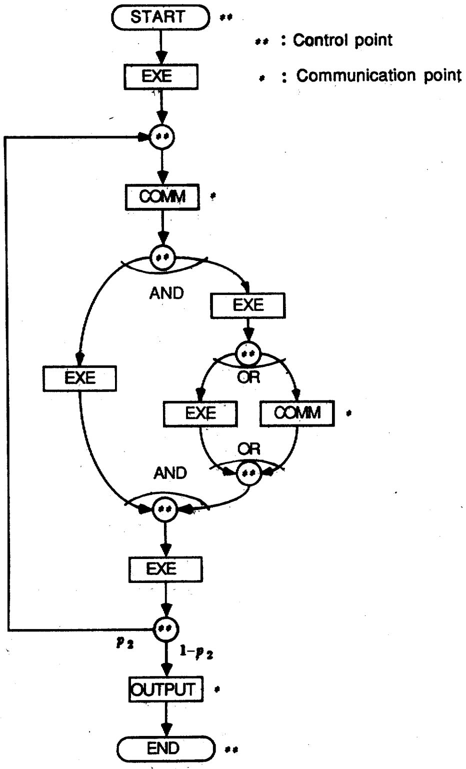


Fig. 16. The CFG of the TFG in Fig. 15.

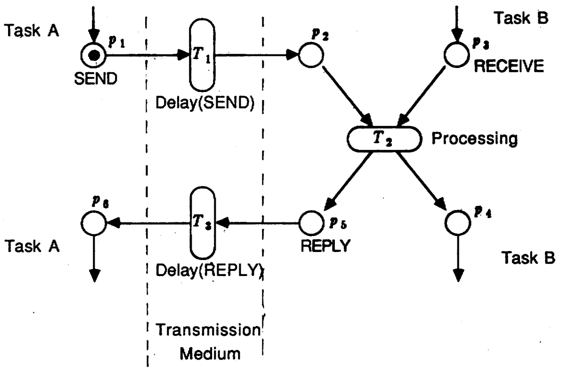


Fig. 17. GSPN model for SEND-RECEIVE-REPLY.

they are typical to real-time control systems. If task A issues a SEND to task B, A remains blocked until a REPLY message from B is received. If B executes a RECEIVE before the message arrives, it also remains blocked. QUERY and RESPONSE are used to allow one task to interrupt another for information, e.g., to avoid collision between two robots that share the same workspace. If A issues a QUERY to B, A remains blocked until the RESPONSE message from B arrives. Upon arrival of the QUERY message from A, B can decide to either accept the QUERY and respond to A immediately or queue the QUERY for a later RESPONSE. WAITFOR is the primitive to allow more than two tasks to synchronize among themselves. Note that a processor can switch to other tasks while the current task is being blocked.

a) SEND-RECEIVE-REPLY: Assuming task A issues a SEND to task B in a different node, the GSPN model for this communication is given in Fig. 17.

When A issues a SEND, a token will be placed in p_1, and a timed transition T_1, which represents the transmission delay of the message, is fired. A token will be created at p_2 at the other end of the message transmission after t_1 units of time, the time required for T_1. A token will also be placed in p_3 if B executes a RECEIVE. This token together with that in p_2 will enable the timed transition T_2, which represents the necessary processing by B after the message is received. Upon completion of T_2, two tokens are generated. One is placed in p_4 to unblock B, the other in p_5 to issue a REPLY to unblock A. Again, the delay for the REPLY message is represented by the timed transition T_3. If A and B are executed on the same node, then the message passing delays T_1 and T_3 are zero, implying that p_1 and p_6 will coincide with p_2 and p_5, respectively.

b) QUERY-RESPONSE: QUERY is used to allow a task to interrupt another task for information. The queried task will decide to either accept or queue the QUERY. When the task accepts the QUERY, it stops its current thread of control, starts executing the RESPONSE routine, returns to where it left off after the RESPONSE routine is completed, and issues a REPLY to unblock the querying task. The GSPN model for QUERY is shown in Fig. 18. In essence, the task issuing a QUERY creates an activity, i.e., the RESPONSE routine, to be executed by the task accepting the QUERY. Upon arrival of the QUERY message, the corresponding RESPONSE routine will be ready to be scheduled for execution by the accepting task. Note that such a GSPN allows a blocked task to respond to or queue a query, and to schedule RESPONSE's in case of multiple queries.

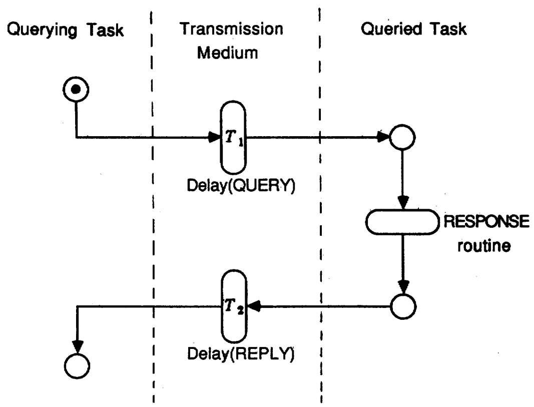


Fig. 18. GSPN model for QUERY-RESPONSE.

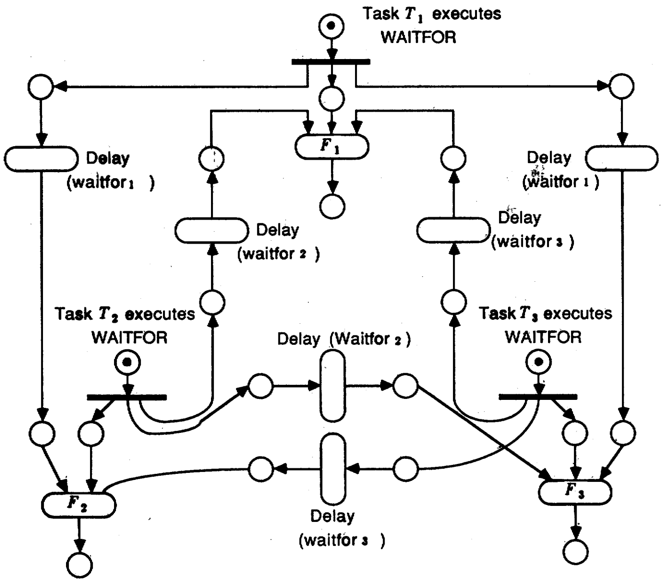


Fig. 19. Three tasks executing WAITFOR's.

c) WAITFOR: WAITFOR allows more than two tasks to synchronize with one another. It can be implemented as follows. When a task T_i in a node executes a WAITFOR, it sends a message, say *waitfor*$_i$, to each of the tasks named in its WAITFOR list. T_i remains blocked until it receives the *waitfor* messages from all the tasks in its waitfor list. After T_i is unblocked, a named function F_i included in the WAITFOR will be executed. When F_i is completed, execution of the task continues from the point immediately after the WAITFOR. Fig. 19 is the GSPN model of WAITFOR for three tasks residing at different nodes participating in a three way synchronization.

After precedence constraints F1) and F2) are properly modeled, a system-wide GSPN can be obtained by "pasting together" all GSPN's corresponding to each individual task invocation in a planning cycle. The system-wide GSPN is unmarked until tasks are invoked. Section IV-B-2) describes how the unmarked GSPN is marked at each task invocation to correctly model the behavior of the task system.

2) Modeling the Time-Driven Task Invocations: Since Petri net-type modeling bases are good only for event-driven parallel transition firings, time-driven task invocations add more complexities to our modeling process. This is because time-driven task invocations dictate the state evolution of the (marked) GSPN at invocation times. Therefore, instead of a single CTMC, a sequence of CTMC's has to be used to correctly model the task system.

Assume, as before, that each periodic task begins its first invocation at time t_0, and that the set of invocation times in a planning cycle $[t_0, t_0 + L)$ is $\{w_1, w_2, \cdots, w_l\}$ where $0 \le t_0 = w_1 < w_2 < \cdots < w_l < t_0 + L$. The following steps show how the unmarked system-wide GSPN built above is marked such that a sequence of CTMC's, $\{(S_k, \Lambda_k, \Theta_k) | k = 1, 2, \cdots, l\}$, can be built for the task system.

1) At the beginning of a planning cycle w_1, the unmarked system-wide GSPN is marked by creating a token in each place corresponding to the START points of the individual GSPN's of all periodic tasks to serve as the initial marking of the planning cycle.

2) The marking at time $t \in [w_1, w_2)$ is determined by the event-driven transition firings in the system-wide GSPN with its initial marking at time w_1.

3) The marking at time w_j, $2 \le j \le l$, is determined by three parts: i) a token is created in each place corresponding to the START points of the individual GSPN's of the tasks invoked, ii) to disallow task pipelining, all tokens associated with previous task invocations are removed from the individual GSPN's before the same tasks are invoked again, and iii) the number of tokens in each place is determined by the event-driven transition firings of the marked system-wide GSPN with its initial marking at time w_{j-1}. This marking serves as the initial marking of the GSPN beginning at time w_j.

4) At time $t \in [w_j, w_{j+1})$, $2 \le j \le l$ where $w_{l+1} = t_0 + L$, the marking is determined by the event-driven transition firings in the system-wide GSPN with its initial marking at time w_j.

For example, the unmarked GSPN of a task system that consists of three periodic tasks T_1, T_2, and T_3 residing in three different nodes A, B, and C is shown in Fig. 20, where T_1 queries T_2 for information and T_3 communicates with T_2 by sending messages. T_1, T_2, and T_3 with periods 5, 10, and 5, respectively, are first invoked at time $t_0 = 0$. In Fig. 20, immediate transitions are denoted by "bars," timed transitions by "ovals," and integer numbers in circles are the place numbers. At the beginning of the planning cycle, a token is generated in p_1, p_9, and p_{18}; at time $t \in [0, 5)$, the state evolution is driven by event-driven transition firings. At $t = 5$ when T_1 and T_3 are invoked again, the marking of the GSPN is determined by: i) a token is generated in p_5 and p_{23} to indicate the second invocations of T_1 and T_3, ii) all tokens in p_1-p_4 and p_{18}-p_{22} are removed to disallow task pipelining, and iii) the tokens in p_9-p_{17} are determined simply by event-driven transition firings since $t = 0$. Likewise, at $t \in [5, 10)$, the state evolution is determined by event-driven transition firings. At $t = 10$, the same is repeated again for the next planning cycle. Notice that the random switches at p_{13} and p_{21} should be the same to assure fault-free message passing since no time-out provisions are made in receiving messages. The same also holds for the two random switches at p_{15} and p_{26}.

Following the execution rules of GSPN introduced in Section II, it is easy to prove that if the task system is fault-free, then the system-wide GSPN marked above is safe.

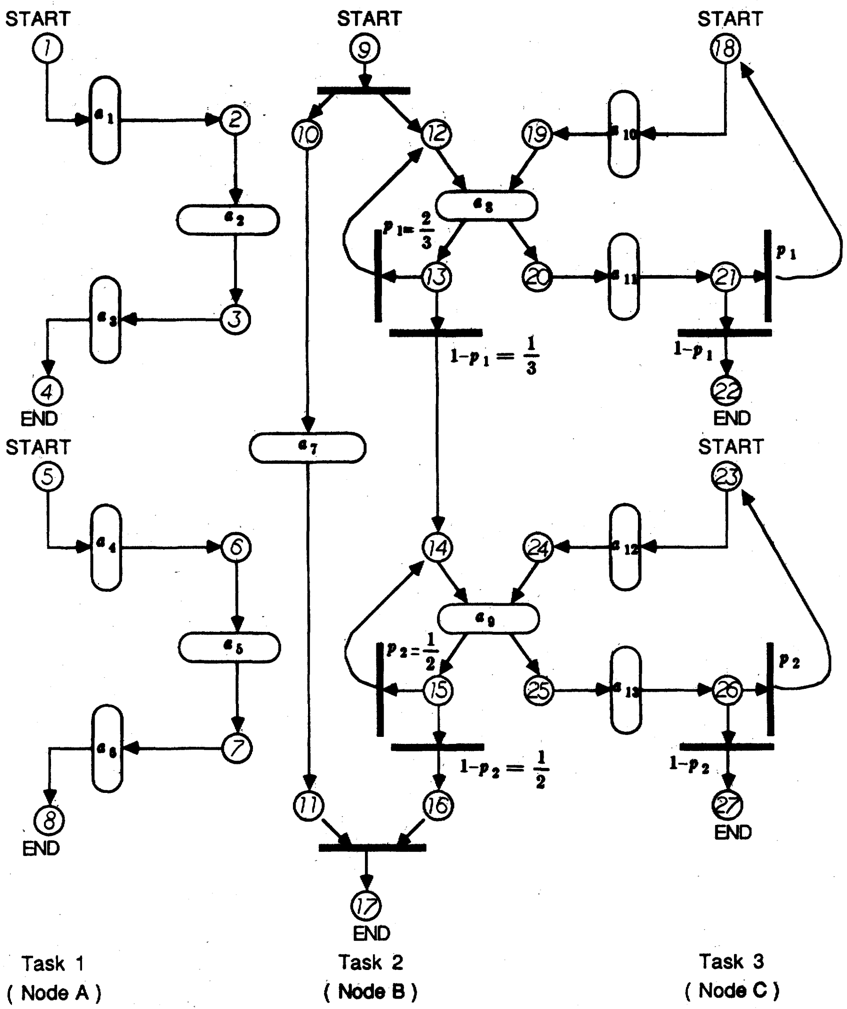


Fig. 20. System-wide GSPN for an example task system.

C. Construction of the CTMC Model

Given the system-wide GSPN marked as above, the state-transition-rate (STR) diagram of the kth CTMC is built as follows:

1) For the ease of analysis, replace each random switch with its GSPN equivalent [10] as shown in Fig. 21.

2) Find S_k by performing reachability analysis on the GSPN initially marked at w_k. This set serves as the state space of the STR diagram in $[w_k, w_{k+1})$. Note that, except those introduced above, there are no vanishing states in S_k.

3) Assume that each timed transition delay between states is independently and exponentially distributed with a transition rate equal to the reciprocal of the expected execution time of the corresponding activity.

For example, the STR diagrams of the task system in Fig. 20 are given in Fig. 22 with $\lambda_1 = \lambda_3 = \lambda_4 = \lambda_6 = 2$, $\lambda_2 = \lambda_5 = 4$, $\lambda_7 = \lambda_{10} = \lambda_{11} = \lambda_{12} = \lambda_{13} = 1$, $\lambda_8 = 6$, $\lambda_9 = 4$, and branching probabilities $p_1 = 2/3$, and $p_2 = 1/2$ where λ_i is the transition rate for activity a_i.

As shown in Fig. 22(a) and (b), all states are represented by a set of integers representing the set of (noninstantaneous) places each with one token, and the number on each arc indicates the transition rate.

Examination of these diagrams leads to the following insights.

- Each state in the *total* state space $S = \cup^l S_i$ globally

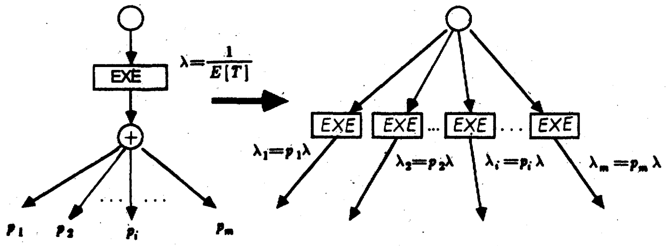
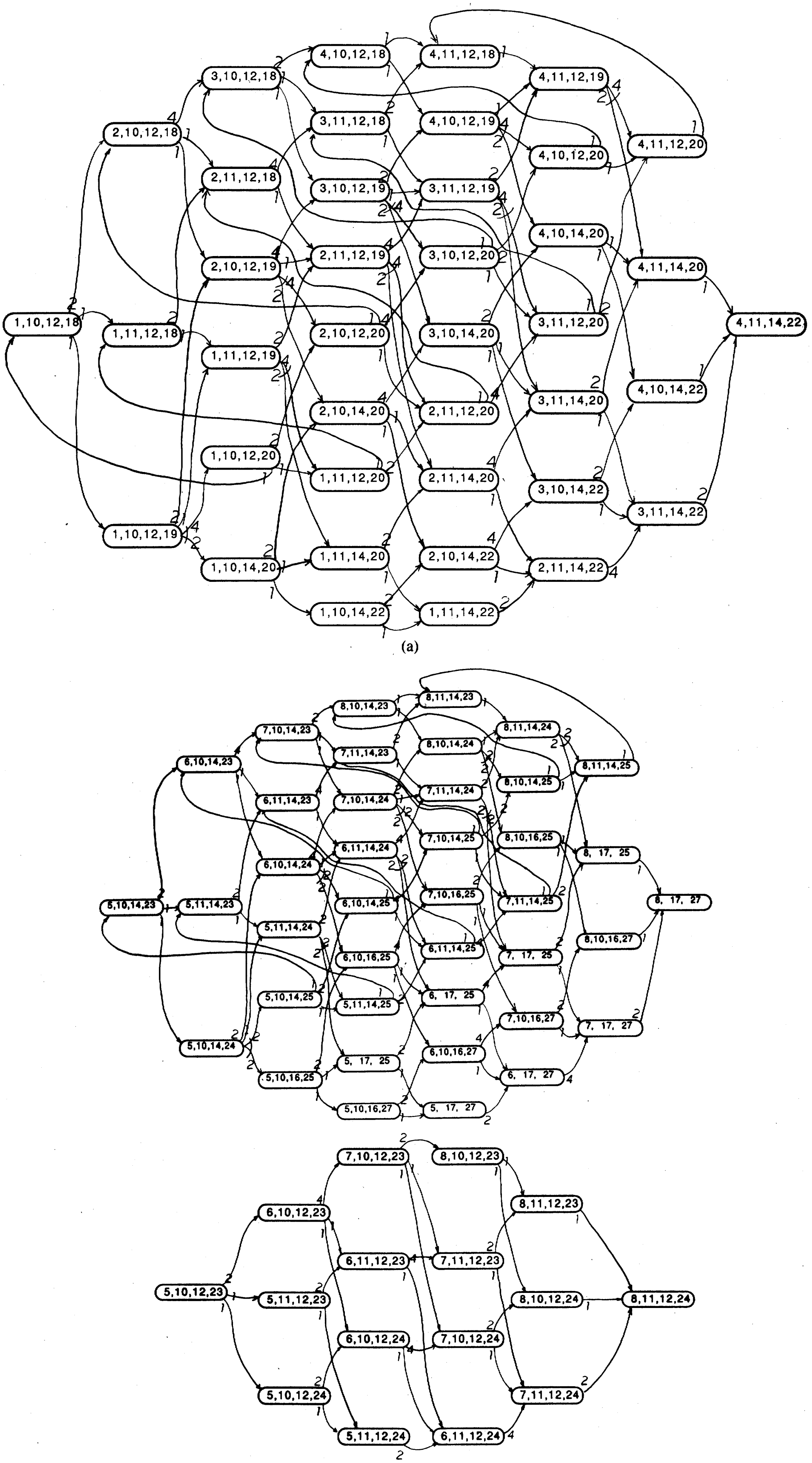


Fig. 21. The GSPN equivalent of a random switch.

describes which stage of the tasks each node has been executing. For example, (2, 10, 12, 19) in Fig. 22(a) represents a state where T_1 is blocked waiting for a response from T_2, T_2 has neither finished a_7 nor responded to the query from T_1, and the message from T_3 has arrived at B waiting to be handled.

- The state spaces for any two different invocation intervals are disjoint. Further, S_1 contains a unique *starting state* s_0, and S_l contains a unique *ending state* s_f of the planning cycle. For example, state (1, 10, 12, 18) (Fig. 22(a)) is the starting state, while state (8, 17, 27) (Fig. 22(b)) is the ending state.
- If a CFG contains loops, then all STR diagrams will be cyclic as shown in Fig. 22(a) and (b), and acyclic otherwise.
- Depending on the marking at w_j, the STR diagram in $[w_j, w_{j+1})$ could be disconnected (Fig. 22(b)).
- Since the precedence constraints among all activities are



(a)

(b)

Fig. 22. The CTMC model for the task system in Fig. 20. (a) $t \in [0, 5)$. (b) $t \in [5, 10)$.

properly conveyed in the system-wide GSPN, the resulting STR diagrams implicitly show all possible execution sequences of activities of the task system in a planning cycle.

- Depending on the task system, some states could be *time critical*. For example, if there is a hard deadline for node A to complete the first invocation of T_1, then at least one state of the form $(4, *, *, *)$ in Fig. 22(a) has to be reached before that deadline.
- For a specific state, the number of activities that can be concurrently executed by a node may be larger than that of the processors available to that node. For example, the system is in state (6, 10, 14, 24), three activities (a_5, a_7, and a_9) can be concurrently executed by node B. If B has only two processors, then a decision must be made as to which two of the three activities in B are to be chosen for parallel execution.

V. An Application Example

In this section, we demonstrate the use of the proposed CTMC model $\{(S_k, \Lambda_k, \Theta_k) | k = 1, 2, \cdots, l\}$ by computing the probability of missing a hard deadline given the activity selection policy and the local state of a node. Clearly, estimating task execution time is a special case of this problem.

The concurrent task execution can be thought as the "movement" of tokens in the GSPN. A place in the GSPN is said to be *realized* if it has a token implying the completion of those activities that "precede" the place. Therefore, a hard deadline in the task system is associated with the realization of a place. For example, a hard deadline for node A in Fig. 20(a) to complete the first invocation of T_1 is the maximum time allowed for p_4 to be realized. A place is said to be *time critical* if it has a hard deadline. A state s_i which contains a set, say R_i, of realized time critical places is called a *goal state with the realized set* R_i, and written as $\psi(s_i) = R_i$. Denote the set of all time critical places by Ψ. Clearly, $\psi(s_i) \subset \Psi$, $\forall s_i \in S$. For the previous example, if $\Psi = \{4, 8, 17\}$, then $\psi((4, 11, 14, 22)) = \{4\}$, and $\psi((6, 17, 25)) = \psi((7, 17, 27)) = \{17\}$. If $\Psi = \{11, 22, 27\}$, then $\psi((6, 11, 14, 25)) = \{11\}$, $\psi((8, 17, 27)) = \{11, 27\}$,[10] and $\psi((8, 10, 16, 25)) = \emptyset$. The set G_h of goal states each of whose realized sets contains the time critical place p_h is called the *goal set with respect to place* p_h, and is written as $\Upsilon(p_h) = G_h$. Obviously, $\Upsilon(p_h) \subset S$, $\forall p_h \in \Psi$. For the same task system in Fig. 20, $\Upsilon(p_{17})$ is the subset of all states of the form $(*, 17, *)$.

The *first passage time* $Y_{i,j}$ from s_i to s_j is an r.v. representing the time needed for the task system to reach s_j for the first time from s_i. Denote the CDF of $Y_{i,j}$ by $Q_{i,j}(t) = \Pr\{Y_{i,j} \le t\}$. A set of initial states I with known element probabilities could sometimes be given instead of a single initial state s_i. Similarly, a set of final states F may be known in place of a single final state s_j. In such a case, the first passage time is defined as the time needed for the task system to reach any $s_j \in F$ for the first time given that the task system is initially in some state $s_i \in I$ at time t_c. Denote the CDF of the first passage time of this type by $Q_{I,F}(t_c, t')$ where t' represents the time measured from t_c.

The local state space S_k^m associated with node m within $I_k = [w_k, w_{k+1})$ is constructed as follows: identify the set of places P_m belonging to node m, and then select the markings of P_m from S_k. Note that for any $h \ne k$, $S_h^m \cap S_k^m = \emptyset$ because $S_h \cap S_k = \emptyset$. Taking the task system in Fig. 20 again as an example, p_1, p_4, p_5, and p_8 belong to node A, $S_1^A = \{(1), (4), (\text{nil})\}$ and $S_2^A = \{(5), (8), (\text{nil})\}$ where (1), (4), (5), and (8) stand for the local states that each of p_1, p_4, p_5, and p_8 contains a token, and (nil) for the local state where no place contains a token. Clearly, $S_k \subset S_k^1 \times S_k^2 \times \cdots \times S_k^m \times \cdots \times S_k^c$, $\forall k = 1, 2, \cdots, l$ where $c = |M|$.

Without loss of generality, we can assume that the current planning cycle starts at time 0. Given the activity selection policy δ^* used by the task system and the local state x^m at time $t_c \ge 0$, we want to compute the probability σ_k of missing the hard deadline $\tau_k \ge t_c$ for a time critical place p_k in the current planning cycle. More formally, we want to calculate

$$\sigma_k = \Pr\{T_k > \tau_k | \delta = \delta^*, X^m(t_c) = x^m\} \tag{4}$$

where T_k is an r.v. representing the time p_k is realized for the first time, δ a variable representing the activity selection policy used by the task system, and $X^m(t)$ an r.v. for the local state of node m at time t.

Given the GSPN and CTMC models of the task system, the above problem can be solved by using the following CTMC properties.

S1) Construct the CTMC model that includes the activity scheduling policy δ^*. This is done by deleting all arcs not in the active transition set $\text{ATS}_i^{\delta^*}$ from the set $\text{OUT}(s_i, *)$ for each $s_i \in S$ to queue the corresponding activities for later execution.

S2) Solve the forward Chapman-Kolmogorov (C-K) equations for the sequence of CTMC's from S1) to obtain the prior probability $P_i(t_c)$ and apply Bayes' theorem [13] to get the posterior probability $P_i'(t_c)$ that the system is in s_i at time t_c given $X^m(t_c) = x^m$. The initial probability of state s_b of the CTMC for interval $I_k = [w_k, w_{k+1})$ is determined from the final probability of state s_a of the CTMC for $I_{k-1} = [w_{k-1}, w_k)$, $k \ne 1$, by the following equation:

$$P_b(w_k) = \begin{cases} 0 & \text{if } B = \emptyset \\ \sum_{s_a \in B} P_a(\omega_k) & \text{otherwise} \end{cases} \tag{5}$$

where $B = \{s_a | \Theta_{k-1}(s_a) = s_b\}$. By Bayes' theorem, the posterior probability is

$$P_i'(t_c) = \Pr\{X(t_c) = s_i | X^m(t_c) = x^m\}$$
$$= \begin{cases} \dfrac{P_i(t_c)}{\sum_{s_j \in L_{x^m}} P_j(t_c)} & \text{if } s_i \in L_{x^m} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

where L_{x^m} is the set of states with $X^m(t_c) = x^m$.

S3) Identify the goal set $\Upsilon(p_k) = G_k$. Since each $s_j \in G_k$ is a state with p_k realized, the event that p_k is realized is equivalent to the event that at least one state in G_k is reached.

S4) Using the posterior probabilities $P_i'(t_c)$ computed in S2) as the initial probabilities at time t_c, compute the CDF of

[10] The realized set of (8, 17, 27) is that of (8, 11, 16, 27).

the first passage time $Q_{L_{x^m},G_k}(t_c, t')$ to any s_j in G_k given that $X^m(t_c) = x^m$ where t' is the time period measured from t_c. $Q_{L_{x^m},G_k}(t_c, t')$ can be found by making each state in G_k an absorption state in the CTMC model, and then solving the C–K equations again for these new CTMC's to compute $\Sigma_{s_j \in G_k} \Pr\{X(t') = s_j | X^m(t_c) = x^m\}$.

S5) Considering the current time t_c and $Q_{L_{x^m},G_k}(t_c, t')$ computed above, σ_k is determined by

$$\sigma_k = \Pr\{T_k > \tau_k | \delta = \delta^*, X^m(t_c) = x^m\}$$
$$= 1 - Q_{L_{x^m},G_k}(t_c, \tau_k - t_c). \qquad (7)$$

As an example, each node A, B, or C in Fig. 20 is assumed to have only one processor dedicating to normal computations. From Fig. 20 or 22, it is easy to see that B is the only node with an insufficient number of processors since, among the set $\{a_2, a_7, a_8\}$ or $\{a_5, a_7, a_9\}$, two or more transitions can be fired simultaneously at some state. Thus, the activity selection policy is determined by how B picks its transition to fire at each state. In this example, we assume that B picks the transition according to the order a_2, a_7, a_8 or a_5, a_7, a_9 to fire in each state. Fig. 23 is the CTMC's corresponding to this policy.

Suppose the only information available to derive σ_k is that p_1 is realized in [0, 5) and p_5 is realized in [5, 10). Suppose also that p_4, p_8, p_{17}, p_{22} and p_{27} are all time critical with hard deadlines $\tau_4 = \tau_{22} = 5$, $\tau_8 = \tau_{17} = \tau_{27} = 10$. Following the above solution steps, we obtain Fig. 24 showing the probabilities of missing these hard deadlines as functions of t_c, the time the information is observed. It is not surprising that σ_4 and σ_8 are identical, since B always chooses a_2 or a_5 first, whenever possible. This implies that the ordered activities $\{a_1, a_2, a_3\}$ or $\{a_4, a_5, a_6\}$ will be executed without interruption. For $t_c \in [0, 5)$ and $t_c \in [5, 10)$, σ_{17} is nearly stable at the value 0.575 because of the dominating fact that, after the second invocation of T_3, T_2 may be waiting hopelessly at p_{12} for the message which will never arrive from the first invocation of T_3 that has already been discarded. σ_{22} is a monotonically decreasing, although not significant, function of t_c. The high probability of missing τ_{22} is due to the high branching back probability p_1 as well as the tightness of the deadline. σ_{27} fluctuates around 0.628, a value larger than σ_{22} by approximately 0.1 because, in addition to similar reasons for σ_{22}, the message from the current invocation of T_3 may wait hopelessly to be processed by T_2 which, unfortunately as discussed above, is also waiting hopelessly for the message from the already-discarded invocation of T_3. From this example, we can see the importance of the time-out mechanism on message communications in a distributed real-time system.

VI. Conclusion

A CTMC model is presented in this paper to model the concurrent task execution in a distributed real-time system. In the modeling process, activities are first identified by alternately applying combination and expansion phases on the original TFG's of the task system. Secondly, the activities and the precedence constraints among them are modeled by a system-wide GSPN. Finally, after considering time-driven task invocations, a sequence of CTMC's is built with the assumption of independently, exponentially distributed activity execution times.

The proposed model has high potential use for resolving various design issues of distributed real-time systems, such as task execution time estimation, message handling, time-out, and task allocation. These issues will be treated in our follow-up papers.

Appendix

Denote the upper and lower endpoints of a chain,[11] the fork and join points of an And-Subgraph, the fork and join points of an Or-Subgraph, and the collecting and branching points of a loop by $C1$, $C2$, $A1$, $A2$, $R1$, $R2$, $L1$, and $L2$, respectively.

There are a total of 39 cases to be considered, each of which is in the form of $P1 \rightarrow P2$. The aggregation rule to reduce the number of redundant structure points for each case is formulated in the following format:

$P1 \rightarrow P2$: AGGREGATION RULE,

CONTROL POINT ASSIGNMENT RULE.

The AGGREGATION RULE indicates either of the following two actions:

a) $G \Rightarrow x$ means "aggregate $P1$ and $P2$, and replace them by the structure point x" where x is either $P1$, $P2$, or both.

b) *NG* means "do not aggregate."

The CONTROL POINT ASSIGNMENT RULE also indicates either of the following two actions:

a) $S \Rightarrow y$ means "assign a control point to structure point y" where y is either $P1$ or $P2$.

b) NS means "no control point is assigned."

The following are the rules for all 39 cases.

CL1: P2 is in the Active Layer, Which is One Layer Inside the Layer of P1

(1) $C1 \rightarrow A1$: $G = > A1$, *NS*

(2) $A1 \rightarrow C1$: $G = > A1$, *NS*

(3) $C1 \rightarrow R1$: $G = > R1$, *NS*

(4) $R1 \rightarrow C1$: $G = > R1$, *NS*

(5) $C1 \rightarrow L1$: $G = > L1$, *NS*

(6) $L1 \rightarrow C1$: $G = > L1$, *NS*

(7) $A1 \rightarrow A1$: $G = > A1$ (outer layer), *NS*

(8) $R1 \rightarrow A1$: *NG*, $S = > A1$

(9) $A1 \rightarrow R1$: *NG*, $S = > R1$

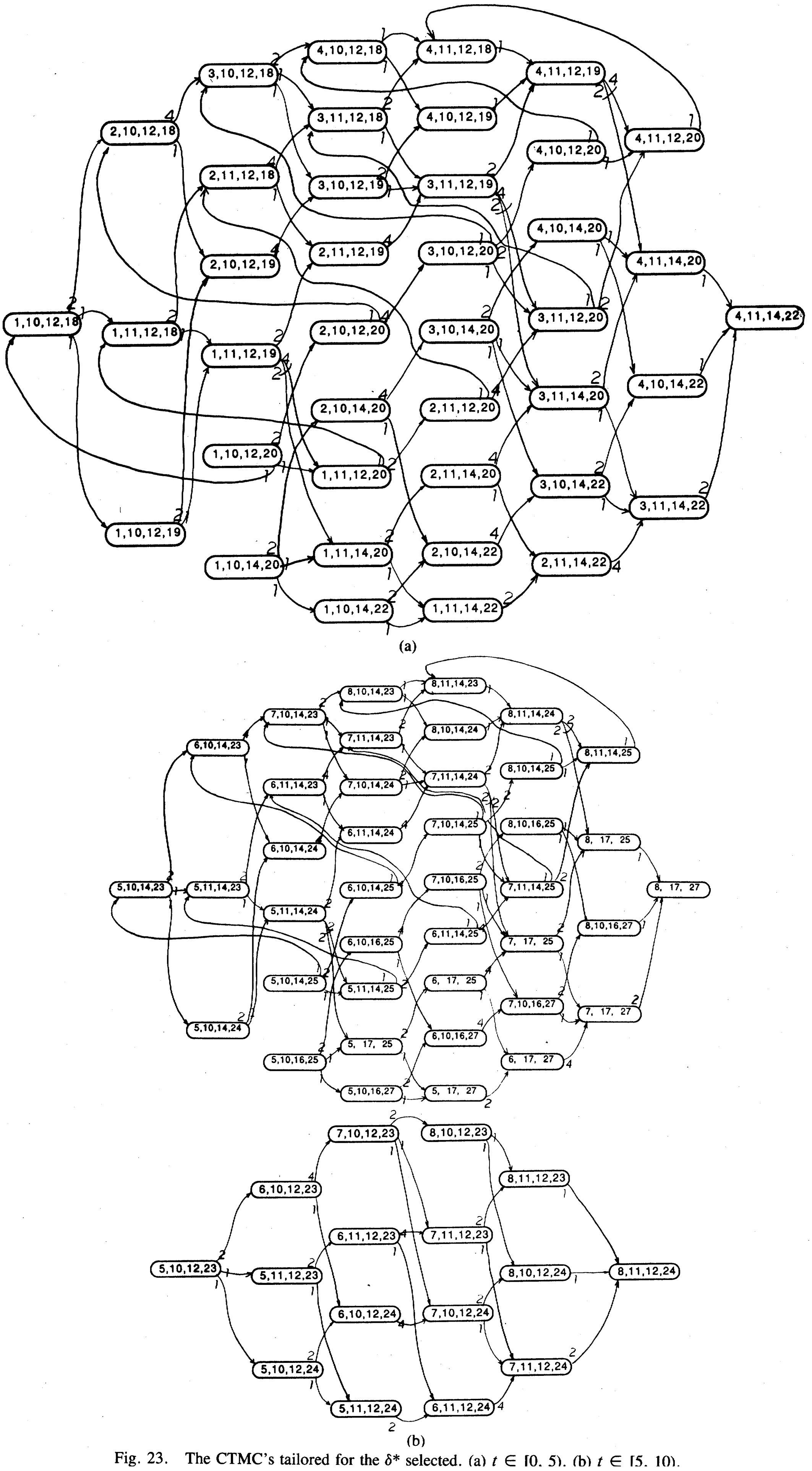
(10) $L1 \rightarrow A1$: *NG*, $S = > A1$

(11) $A1 \rightarrow L1$: *NG*, $S = > L1$

(12) $R1 \rightarrow R1$: $G = > R1$ (outer layer), *NS*

(with proper adjustments of the

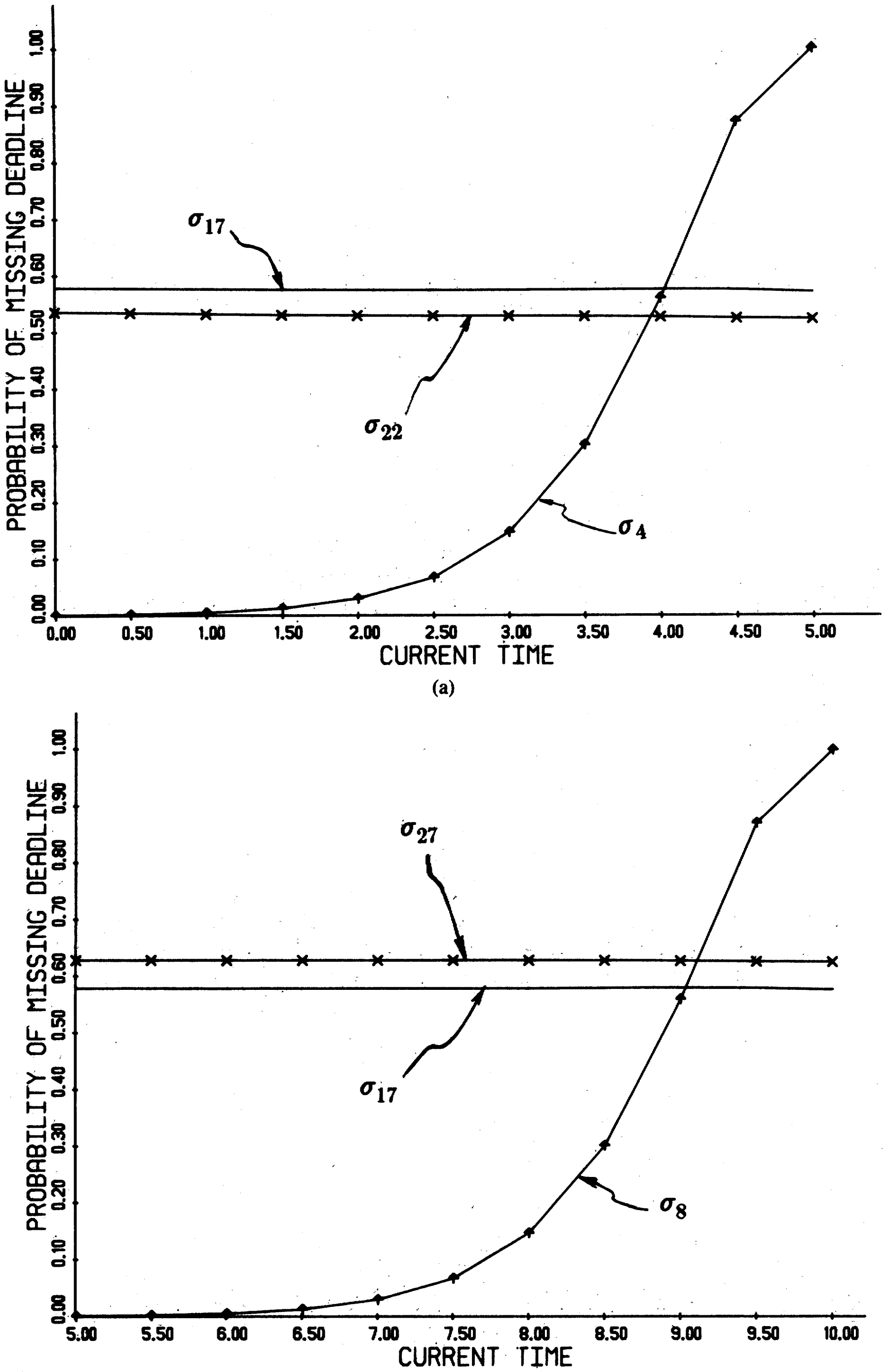
[11] The START and END points of the TFG are treated as upper and lower endpoints, respectively.



(a)

(b)

Fig. 23. The CTMC's tailored for the δ^* selected. (a) $t \in [0, 5)$. (b) $t \in [5, 10)$.



(b)

Fig. 24. The probabilities of missing deadlines. (a) $t_c \in [0, 5)$. (b) $t_c \in [5, 10)$.

branching probabilities of the inner layer Or-Subgraph.)

(13) $L1 \rightarrow R1$: NG, $S = > R1$

(14) $R1 \rightarrow L1$: NG, $S = > L1$

(15) $L1 \rightarrow L1$: $G = > L1$ (outer layer), NS.

CL2: P1 is in the Active Layer, Which is One Layer Inside the Layer of P2

(16) $C2 \rightarrow A2$: $G = > A2$, NS

(17) $A2 \rightarrow C2$: $G = > A2$, NS

(18) $C2 \rightarrow R2$: $G = > R2$, NS

(19) $R2 \rightarrow C2$: $G = > R2$, NS

(20) $C2 \rightarrow L2$: $G = > L2$, NS

(21) $L2 \rightarrow C2$: $G = > L2$, NS

(22) $A2 \rightarrow A2$: $G = > A2$ (outer layer), NS

(23) $A2 \rightarrow R2$: NG, $S = > A2$

(24) $R2 \rightarrow A2$: NG, $S = > R2$

(25) $A2 \rightarrow L2$: NG, $S = > A2$

(26) $L2 \rightarrow A2$: NG, $S = > L2$

(27) $R2 \rightarrow R2$: $G = > R2$ (outer layer), NS

(28) $R2 \rightarrow L2$: NG, $S = > R2$

(29) $L2 \rightarrow R2$: NG, $S = > L2$

(30) $L2 \rightarrow L2$: $G = > L2$ (outer layer), NS (with proper adjustments of the branching probabilities of the outer layer loop).

CL3: Both P1 and P2 are in the Active Layer

(31) $A2 \rightarrow A1$: $G = > A2$ (or $A1$), $S = > A2$ (or $A1$)

(32) $A2 \rightarrow R1$: NG, $S = > A2$ and $R1$

(33) $A2 \rightarrow L1$: NG, $S = > A2$ and $L1$

(34) $R2 \rightarrow A1$: NG, $S = > R2$ and $A1$

(35) $R2 \rightarrow R1$: NG, $S = > R2$ and $R1$

(36) $R2 \rightarrow L1$: NG, $S = > R2$ and $L1$

(37) $L2 \rightarrow A1$: NG, $S = > L2$ and $A1$

(38) $L2 \rightarrow R1$: NG, $S = > L2$ and $R1$

(39) $L2 \rightarrow L1$: NG, $S = > L2$ and $L1$.

Acknowledgment

The authors wish to thank M. Woodbury of the University of Michigan and anonymous referees for their constructive comments on an earlier version of this paper.

References

[1] K. G. Shin, C. M. Krishna, and Y.-H. Lee, "A unified method for evaluating real-time computer controllers and its application," *IEEE Trans. Automat. Contr.*, vol. AC-30, pp. 357–366, Apr. 1985.

[2] J. P. Huang, "Modeling of software partition for distributed real-time applications," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1113–1126, Oct. 1985.

[3] M. L. Virginia, "Task assignment to minimize computation time," in *Proc. 1985 IEEE Int. Conf. Distrib. Comput. Syst.*, pp. 329–336.

[4] A. K. Mok, "Fundamental design problems of distributed systems for the hard real-time environment," Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, May 1983, pp. 80–94.

[5] W. W. Chu and K. K. Leung, "Task response time model & its applications for real-time distributed processing systems," in *Proc. IEEE 1984 Real-Time Syst. Symp.*, pp. 225–236.

[6] M. Ajmone Marsan, G. Balbo, and G. Conte, "A class of generalized stochastic Petri nets for the performance analysis of multiprocessor systems," *ACM TOCS*, vol. 2, pp. 93–122, May 1984.

[7] L. Kleinrock, *Queueing Systems, Volume I: Theory.* New York: Wiley, pp. 44–53, 1975.

[8] S. M. Ross, *Introduction To Probability Models.* New York: Academic, 1983.

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

[10] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks," Commun. Network Lab., Industrial Technology Inst., Ann Arbor, MI, Tech. Rep., 1984.

[11] M. K. Molloy, "On the integration of delay and throughput measures in distributed processing models," Ph.D. dissertation, Univ. California, Los Angeles, 1981, pp. 19–21 and 53–54.

[12] K. G. Shin and M. E. Epstein, "Intertask communications in an integrated multi-robot system," in *Proc. 1985 IEEE Conf. Robotics and Automation*, pp. 910–917; also to appear in *IEEE J. Robotics and Automation.*

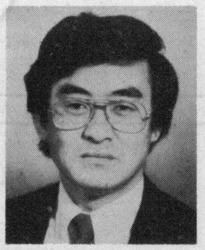
[13] M. H. DeGroot, *Optimal Statistical Decisions.* New York: McGraw-Hill, 1970, pp. 11–12.

[14] W. Feller, *An Introduction to Probability Theory and Its Applications, Vol. 2.* New York: Wiley, 1971, pp. 491–495.

Dartzen Peng (S'85) received the B.E.E. degree from National Cheng Kung University, Tainan in 1974, and the M.S. degree in Management Science from National Chiao Tung University, Hsinchu in 1976, both in Taiwan, Republic of China.

He is currently working towards the Ph.D. degree in the CSE Division of the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor. From 1978 to 1982, he was with the Computer Technology Development Center, ERSO/ITRI in Taiwan. Since 1985 he has been a Research Assistant at the University of Michigan. His research interests include computer networks, distributed real-time computing, integrated CAD and CAM systems, and MIS.

Mr. Peng is a student member of the IEEE Computer Society.



Kang G. Shin (S'75–M'78–SM'83) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and both the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, in 1976 and 1978, respectively.

From 1970 to 1972 he served in the Korean Army as an ROTC officer and from 1972 to 1974 he was on the research staff of the Korea Institute of Science and Technology, Seoul, working on the design of VHF/UHF communication systems. From 1974 to 1978 he was a Teaching/Research Assistant and then an Instructor in the School of Electrical Engineering, Cornell University. From 1978 to 1982 he was an Assistant Professor at Rensselaer Polytechnic Institute, Troy, NY. He was also a Visiting Scientist at the U.S. Air Force Flight Dynamics Laboratory in summer 1979 and at Bell Laboratories, Holmdel, NJ, in Summer 1980 where his work was concerned with distributed airborne computing and cache memory architecture, respectively. He also taught short courses for the IBM Computer Science Series in the area of computer architecture. Since September 1982, he has been with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, where he is currently an Associate Professor. His current teaching and research interests are in the areas of distributed and fault-tolerant computing, computer architecture, and robotics and automation. He was the Program Chairman of the 1986 IEEE Real-Time Systems Symposium and is the Guest Editor of the Special Issue of IEEE TRANSACTIONS ON COMPUTERS on Real-Time Systems which is scheduled to appear in August 1987. He has been very active and has authored or coauthored over 100 technical papers in the areas of distributed fault-tolerant real-time computing, computer architecture, and robotics and automation. As an initial phase of validation of architectures and analytic results, he and his students are currently building a 19-node hexagonal mesh at the Real-Time Computing Laboratory (RTCL), University of Michigan.

Dr. Shin is a member of the Association for Computing Machinery, Sigma Xi, and Phi Kappa Phi.