A MODELING METHODOLOGY FOR THE ANALYSIS OF CONCURRENT SYSTEMS AND COMPUTATIONS

Alex Kapelnikov
Richard R. Muntz
Milos D. Ercegovac

July 1987
CSD-870038

A Modeling Methodology for the Analysis of Concurrent Systems and Computations

Alex Kapelnikov+, Richard R. Muntz, and Milos D. Ercegovac

Computer Science Department
University of California, Los Angeles

Los Angeles, California 90024-1600

+ Presently with Unisys, Santa Monica, California.

ABSTRACT

In this paper, we describe a novel modeling methodology for evaluating the performance of distributed, multiple-computer systems. Our approach employs a set of analytic tools to obtain an estimate of the average execution time of a parallel implementation of a program (or transaction) in a parallel environment. These tools are based on an amalgamation of queueing network theory and graph models of program behavior.

I. INTRODUCTION

In developing a distributed system, many design issues must be addressed. Some of the more important of those issues are: the configuration, size, and technology of the interconnection network; distribution of system functionality among its processing units; allocation of workload (program tasks) to processors; how to synchronize execution of tasks; the size of individual tasks; where to store various data sets. A designer of a distributed system is faced with the dilemma of properly resolving these issues in order to meet certain requirements. It is usually not until the final stages of a system's development that it is possible to determine whether or not the original objectives have been met. Thus, an incorrect choice, in deciding on any one of the numerous issues, can result in a very costly and time consuming re-design and re-development effort.

Given the aforementioned considerations, one can appreciate the importance of being able to predict the eventual performance of a system during its design process or primeval stages of development, so that design flaws can be detected and corrected without great expense. With proper performance prediction tools, a designer should be able to gradually "pilot" the design into meeting the specified requirements.

Currently available performance prediction methods for distributed systems fall into two general categories. Methods of the first category employ simulation tools to construct and run a detailed model of a system. Usually general-purpose simulation packages, such as UCLA's SARA System (Graph Model of Behavior) [VER82], IBM's RESQ [SAU81b], or PAWS [INF81], are used, which have built-in facilities for gathering and analyzing performance statistics. Most of such simulation packages are very expensive to use, in terms of their consumption of computer resources and the required model development effort.

The second category of performance analysis tools comprises approximate analytic techniques. Each such technique is generally applicable only to a narrow range of system architectures and specific types of program structures. These methods employ either standard queueing network models or graph models, which trace system states during program execution, e.g., Petri Nets [PET81]. The techniques based only on queueing networks, such as the one proposed in [HEI83], currently suffer from the inability to explicitly represent complex interdependencies of tasks in a program. Such methods usually utilize external, Poisson arrival streams, with "heuristically" chosen customer arrival rates, to represent new tasks being enabled. With most of the techniques based purely on graph models, such as Stochastic Petri Nets [MOL81], the model of program behavior is intimately linked with the model of the execution environment. Thus, a minor architectural change may require a new model to be constructed and solved. Another limitation of most graph-based methods is that their state space size usually grows combinatorially with respect to the size of the particular configuration being modeled.

The preceding discussion shows that currently available performance prediction tools suffer from shortcomings which limit their range of practical application. Specifically, with currently available performance analysis tools, one must sacrifice either computational efficiency, as with general-purpose simulators and some graph-based methods, or generality, as with special-purpose simulators, queueing network models, and other graph-based techniques. Our goal in this research had been to develop a new, general modeling framework for efficiently evaluating the performance of a broad class of distributed, multiple-computer systems. The major premise of our methodology is that the graph-based methods are best suited for modeling precedence relationships between tasks in a program, while techniques based on queueing network models are best suited for representing the details of the execution environment. Therefore, by segregating the model of program behavior from the model of system architecture, we intend to exploit the advantages of both queueing networks and graph models where they are most beneficial. Work along similar lines had been presented in [THO86] and [BAL86]. However, we do believe that our approach has a broader scope of application and offers a number of unique features, which reduce the computational complexity of model construction and analysis.

In the next section, we identify those properties of distributed systems which are pertinent to our methodology and classify such systems accordingly. Sections III and IV describe our modeling framework and the corresponding solution procedure, respectively. In Section V, we present heuristic techniques, based on hierarchical decomposition of program models, for optimizing the solution of several common program types. Section VI illustrates the application of this methodology and Section VII discusses its merits with respect to other performance prediction tools.

II. CLASSIFICATION OF DISTRIBUTED ENVIRONMENTS

In order to make our modeling methodology as computationally attractive as possible without, however, sacrificing its generality, we will classify the general class of distributed, multiple-computer systems according to two pertinent criteria. The type of task allocation policy employed by the system constitutes the first criterion. The second criterion is the way the synchronization of tasks is performed in the system.

Task allocation policies will be classified as either dynamic or static [ACK82, KUC77]. With a dynamic task allocation policy, each enabled task competes, on an equal basis, with other enabled tasks for the processing and communication resources of a system. That is, the required system resources are dynamically assigned to a task (by the task scheduling and resource management components of the architecture) at the time it becomes ready for execution. With a static task allocation policy, each task of a given program is a priori allocated a pre-specified subset of the system's processing and communication resources. Such subsets are not necessarily mutually exclusive nor represent an exhaustive partitioning of the total available system capacity. This allocation of resources is performed before commencing the execution of a program.

The various schemes for synchronizing execution of tasks will be classified as either centralized or distributed. With a centralized synchronization of tasks, all information necessary to determine when each task can become enabled is kept in a single (central) storage facility. Whenever a task completes its execution, it generates a completion acknowledgment or a result packet. Result packets are used to create operand packets which contain the operands needed to enable certain tasks. Since the information about each task is stored in the same location, only a single operand packet has to be generated from a given result packet. With a distributed synchronization of tasks, storage of information about disabled tasks is distributed among a number of memory modules. Each result packet will generate as many operand packets as the number of different storage modules which need to be updated with the information contained in that result packet. The transmission and processing of operand packets generated from the same result packet can be performed asynchronously.

Using the two criteria desribed above, our classification of distributed, multiple-computer systems consists of the four categories listed below:

(1) Dynamic Allocation with Centralized Synchronization

(2) Dynamic Allocation with Distributed Synchronization

(3) Static Allocation with Centralized Synchronization

(4) Static Allocation with Distributed Synchronization

Our objective in distinguishing between these four categories is to avoid as much complexity in our models as much as possible. In fact, the structure of models for representing systems of the fourth group is the most general one and can be used to model systems belonging to other categories as well. However, the models for systems of other classes are less complex and are simpler to construct and solve.

III. THE MODEL

We will now describe how the execution of a program in a distributed environment is modeled by our methodology. The procedure for solving this model will be presented in the following section.

Our methodology embodies two modeling domains. The physical domain represents the physical resources comprising the distributed, multiple- computer system being modeled, such as processors, communication buses, peripheral controllers, I/O device drivers, storage facilities, etc. The program domain comprises programs or processes, each consisting of a set of cooperating tasks. The physical domain model is used to obtain the throughput rates of the system for different types of tasks under various task loadings. These rates are then used to compute the parameters needed for solving the program domain model to obtain an estimate of the average program execution time.

3.1 Physical Domain Model

The most general type of model in this domain consists of a "P/C" (P rocessing and C ommunication) queueing subnetwork, an "M/U" (M atching and U pdating) queueing subnetwork, and two

Black Boxes. (The "Black Boxes" are modeling constructs introduced to represent synchronization constraints related to tasks' interdependencies. A fuller explanation is given shortly.) Figure 3.1 is a high-level representation of such a model.

The P/C subnetwork's service centers represent the processing and communication resources of the system being modeled. The interconnection of the elements, together with the associated routing probabilities, represent the architectural profile of the system. The attributes of each element (e.g., service time distribution of each customer class, number of servers, type of servers, server capacities, queueing discipline used) are determined by the characteristics of the underlying physical resource and the properties of the tasks utilizing that resource. The functions of the portion of the distributed system being modeled by the P/C subnetwork are: (1) receiving executable operand sets and corresponding task definitions from the synchronization subsystem (defined below); (2) transmitting these to the proper processing elements for execution; (3) executing task code; and (4) transmitting the results back to the synchronization subsystem. An operand set, the corresponding task, and the generated result are all modeled by a single queueing network customer (since these are sequential activities).

The service centers of the M/U subnetwork represent delays incurred in contending for, accessing, using, and updating the synchronization subsystem -- the part of the distributed system which is responsible for maintaining the information necessary for task synchronization. The queueing network customers which visit the M/U subnetwork represent operand packets (or "completion acknowledgments" from already executed tasks) which contain information for updating and/or creating task operand sets in certain memory modules. There are as many customer chains in this subnetwork as there are distinct memory modules.

A Black Box is a conceptual artifact designed to model interdependencies between tasks in a program. A queueing network customer which visits the Black Box service center is immediately "destroyed". In turn, some number of customers may be created, depending on the current state of the Black Box. Thus, a Black Box performs the roles of both a source and a sink in a queueing network model. The operations described above are performed instantaneously.

In our physical domain model, Black Box A consumes a customer leaving the P/C subnetwork, which, at that stage, represents a result packet, generated upon completion of execution of some task. It then ejects as many customers into the M/U subnetwork as the number of distinct memory modules which need to be updated with the information contained inside that result packet. Black Box B consumes a customer leaving the M/U subnetwork, which represents an operand packet that has been processed and used to update the contents of some memory module. It then updates descriptors of incomplete operand sets and ejects as many customers into the P/C subnetwork as the number of operand sets which were completed (i.e. number of tasks enabled).

If the system being modeled uses only a single memory module for storing operand sets (i.e., it falls into the "Centralized Synchronization" category), it is not necessary to explicitly include Black Box A in the model. The reason for this is as follows. Since each result packet is always routed to the same, single memory module, for each customer it consumes, Black Box A ejects exactly one customer. Thus, we can eliminate Black Box A and have a customer leaving the P/C subnetwork go directly to the M/U subnetwork. Furthermore, we can then incorporate the service centers of the M/U subnetwork into the P/C subnetwork to avoid having the M/U subnetwork as a separate entity. Each customer in this new, joint subnetwork will be representing, at each time instant, either a task, a result packet or an operand Such model reduction in the physical domain also reduces the complexity of the associated program domain model(s). This, in turn, considerably simplifies the solution process, allows a number of optimizations to be utilized, and permits the application of certain heuristic techniques (see section 5).

3.2 Program Domain Model

A program, as defined in the context of this paper, is a set of tasks, which are executed according to the precedence ordering given by the associated computation control graph (defined below).

3.2.1 Computation Control Graphs

Computation control graphs are a mechanism for pictorially representing program behavior. The particular form of a computation control graph is similar to the already existing graph models of program behavior [DEN72, EST78, FER72, KAR67, MOL81, PET81]. Its novel features include:

- convenient representation of recursive relationships;
- flexibility in modeling looping constructs and multiple instantiations of tasks;
- hierarchical grouping of precedence relationships;
- allowing for automated generation of Markov processes.

Each computation control graph is a collection of nodes and directed arcs connecting those nodes. Nodes represent tasks and arcs model precedence relationships among those tasks. Several arcs can emanate from a single node and a single node can be a destination for multiple arcs. With each node i, we will associate two sets of arcs: E(i) and D(i). E(i) = {s | arc s emanates from node i}. D(i) = {s | arc s terminates at node i}.

The following rules apply to arcs terminating at the same node. If there are several arcs terminating at node i without any symbols between their heads (as shown in Figure 3.2(a)), then the task, represented by node i, cannot be initiated until all of the tasks, represented by the source nodes of those arcs, have completed their execution. This is called the AND relationship and is formally expressed as:

$$D(i) = \{j_1 \bullet j_2 \bullet \ldots \bullet j_n\},$$

where $j_1, j_2, \ldots, j_n$ are the source nodes of the arcs terminating at node i. If those arcs would be joined by "+" symbols (as shown in Figure 3.2(b)), then the task, modeled by node i, could be executed as soon as a task, modeled by any one of the source nodes, had completed its execution. This is called the OR relationship is formally expressed as:

$$D(i) = \{j_1 + j_2 + \cdots + j_n\}.$$

Both AND and OR relationships can be intermixed together, with AND taking precedence over OR. In order to represent more complex precedence relationships, arcs can be grouped hierarchically by drawing ellipses around them (depicted in Figure 3.2(c)). This is called the UNION relationship and is represented by segregating the affected arcs using parentheses. The UNION relationship takes precedence over all other relationships and can be hierarchically applied.

Arcs emanating from the same node are subject to the following interpretation. If there is a weight w (w is a real number between 0 and 1) at an arc's tail (see Figure 3.3(a)), then, upon completion of the task represented by that arc's source node, the arc is enabled with probability w. Arcs emanating from node i, without any symbols between them (as illustrated in Figure 3.3(b)), are joined by the AND relationship, which is formally expressed as:

$$E(i) = \{j_1 / w_1 \bullet j_2 / w_2 \bullet \cdots \bullet j_n / w_n\},$$

where $j_1, j_2, \ldots, j_n$ are the destination nodes for the arcs emanating from node i and $w_1, w_2, \ldots, w_n$ are their respective weights. In the case of arcs joined by the AND relationship, the decision as to whether or not to enable an individual arc is based on that arc's weight alone, i.e., any combination of such arcs can be enabled. If there are "+" symbols between tails of several arcs (as shown in Figure 3.3(c)), then exactly one of those arcs will be enabled. This is called the EXCLUSIVE-OR relationship. The probability that a particular arc is enabled is equal to that arc's weight. This relationship is formally expressed as:

$$E(i) = \{j_1 / w_1 + j_2 / w_2 + \cdots + j_n / w_n\}.$$

EXCLUSIVE-OR takes precedence over AND. Arcs emanating from the same node can also be combined hierarchically by applying the UNION relationship (depicted in Figure 3.3(d)). UNION takes precedence over AND and EXCLUSIVE-OR and can be hierarchically applied.

As an illustration of the rules defined above, let us consider the computation control graph pictured in Figure 3.4. The formal expressions describing the relationships between nodes in this graph are given below:

$E(1) = \{(3|1/2 + 4|1/2)|1/3 \bullet 5|3/4\};$ $D(1) = \{\}$

$E(2) = \{4|5/6 + 5|1/6\};$ $D(2) = \{\}$

$E(3) = \{\};$ $D(3) = \{1\}$

$E(4) = \{\};$ $D(4) = \{1 \bullet 2\}$

$E(5) = \{\};$ $D(4) = \{1 + 2\}$

3.2.2 Classes of Tasks

In order to account for varying demands on the system's physical resources, we can separate tasks into groups, where members of each group have similar processing and communication requirements. The number of groups to use is a subjective decision and depends on the degree of variability in the behavior of tasks, the level of solution accuracy desired, and the bounds on the computational cost of the solution process. Each distinct group of tasks is modeled by a different <u>chain</u> of queueing network customers, in both P/C and M/U subnetworks. For each customer chain, the set of attributes of its customers, such as the service time distribution function at each service center and routing probabilities, are chosen to "most closely" approximate the behavior of members of the corresponding task group in the actual system. In the case of a static task allocation policy, each task group is further divided into smaller groups (subgroups) (according to how system resources were pre-allocated to each task in the original group), such that the tasks in each subgroup are assigned the same subset of resources.

3.2.3 Segmentation: Hierarchical Perspective

It is now appropriate to introduce the concept of a "segment", which is used extensively in section V to describe various approximation techniques. We define a segment to be a set of nodes in a computation control graph which has the following properties. Those nodes in a segment which are destinations for arcs originating from some nodes external to the segment are always all enabled simultaneously. In other words, after some set of nodes in a segment is enabled by external stimuli, the execution of the nodes inside that segment proceeds without any interaction with the nodes outside of the segment. However, all of the precedence relationships among the nodes inside the segment are obeyed. The arcs, which originate from the nodes internal to the segment and terminate at the nodes external to the segment, can be enabled only after all of the nodes in the segment have completed execution. Segments can be viewed as "supernodes" in a "higher level" computation control graph. The rules for interpreting the interconnections of such "supernodes" are the same as those for "ordinary" nodes.

Figure 3.5 illustrates some of the programming constructs (segments) found in typical programs. The sequential construct is formed by combining modules in series. It is representative of programs consisting of a sequence of procedures, and of transactions involving a serial application of several operations to the same data item. In the EXCLUSIVE-OR construct, exactly one out of several possible modules is selected for execution. The i-th module is selected if and only if predicate p_i is satisfied. At each instance, exactly one predicate holds true, while the others are false. This construct is representative of the "Case" statement in programs and of the conditional transaction execution. The parallel construct is formed by combining modules in parallel. It is representative of concurrent transaction processing and of executing independent computations in an algorithm. In the loop construct (Figure 3.5(d)), module A enables itself after enabling module B (thus starting the loop over again) if predicate p is satisfied. This construct is representative of a "DO loop" which repeats itself as long as condition p holds true, where each iteration is a serial combination of modules A and B; however, a new iteration can start as soon as module A of the current iteration is completed. It can also be used to model a recursive procedure, having modules A and B as its body, which calls itself after completing module A (as long as predicate p is satisfied), with module B being independent of the new procedure invocation.

IV. APPROXIMATE ANALYTIC SOLUTION

Our starting point is the description of a program domain model and the description of a physical domain model. With this information, we proceed to:

(1) solve the physical domain model to find system through puts of various customer chains for different states of the system;

(2) construct a Markov process whose state space consists of relevant states of program execution;

(3) approximate state transition rates from the system throughputs; and, finally,

(4) solve the Markov process to obtain an estimate of the average program execution time in the environment being considered.

4.1 Decomposition Approximation

The objective of this approximation is to reduce the computational complexity of our solution procedure by representing the behavior of the physical domain model in a more compact form. The procedure presented here is motivated by Norton's Theorem for decomposing closed queueing network models [LAV82]. The major premise of this theorem states that, if a part of a closed queueing network is replaced by a state-dependent, exponential server with properly chosen service rates and routing transition probabilites into the remainder of the network, then the newly formed network is equivalent, in terms of global performance mesures (e.g., the average total throughput and the average system response time), to the original network [COU77, LAV82, VAN78]. This is graphically demonstrated in Figure 4.1. The service rates are found by constructing a new, smaller closed queueing network from the part of the original network that is being aggregated into one server. This new network is formed by changing all routing transitions to service centers external to the subnetwork into transitions to internal service centers, the particular service centers being determined from the relative frequencies of routing transitions in the original network.

Norton's Theorem is applicable exactly to product-form queueing networks only [LAV82]. Even if both P/C and M/U subnetworks are each product-form, when considered as closed networks individually, the complete queueing network representing a physical domain model is, in general, not product- form. Thus, in applying Norton's Theorem to our "extended" queueing network, we are approximating the behavior of the physical domain model.

Figure 4.2 shows the new physical domain model, which was obtained from the original model (shown in Figure 3.1) by applying the approximation technique described above. In this new, "reduced" model, P/C and M/U are state-dependent, exponential service centers. $n(T_i)$ is the number of chain T_i customers present at the P/C service center -- those are the customers which represent tasks, along with the associated result packets, belonging to group i. $n(O_i)$ is the number of chain O_i customers present at the M/U service center -- those are the customers which represent operand packets destined for memory module i of the synchronization subsystem. Furthermore, we require that each service center employ the Processor-Sharing (PS) scheduling policy within each customer class. This is equivalent to assuming that, when a customer of some class departs from the P/C service center, it is equally likely to be any one of that class.

4.2 Markov Process Construction

In order to find the expected time of execution of a program in the distributed environment being modeled, we need to "track" its behavior during the time period when it is being executed. We accomplish this by constructing a Markov process consisting of all of the "important" states of the program's execution history. One of the key advantages of this methodology is that, for a given program, the structure of the associated Markov process does not depend on a particular execution environment. It only depends on the category the selected physical system belongs to. However, the numerical parameters of that Markov process, such as state transition rates, are functions of the specific system architecture.

4.2.1 State Space Description

Figure 4.3 depicts a state descriptor for the most general category ("Static Allocation with Distributed Synchronization") of the distributed, multiple-computer systems. The part of the descriptor on the left side of the ";" separator specifies all currently "active" queueing network customers, i.e. those representing enabled tasks, result packets, and operand packets. The "T_i Customers" section of this part lists the identities of those enabled tasks which are represented by customers of chain T_i (i.e. the tasks belonging to group i). The "O_i Customers" section of the left part specifies those operand packets that have not yet been processed and which are represented by customers of chain O_i (i.e. the operand packets

destined for memory module i). The "O_i Customers" sections are not needed when modeling systems with centralized synchronization, since, in models of such systems, operand packets are integrated with result packets. The part of the descriptor on the <u>right</u> side of the ";" separator provides additional information by describing all incomplete operand sets present in the synchronization subsystem. The "Incomplete_Operand_Sets$_i$" section, $\{IOS_i\}$, lists all incomplete operand sets stored in memory module i.

A state transition occurs whenever an <u>active</u> customer completes service at either P/C or M/U service center. For a state with a non-empty "T_i Customers" section, whenever a chain T_i customer departs from the P/C service center, a transition takes place into one of several possible states. Each possible transition corresponds to an activation of some "feasible" subset of those arcs in the graph which emanate from the node corresponding to the departing customer. We define a feasible subset of arcs, for a given node in the graph, as a set where all member arcs may be enabled simultaneously upon completion of execution of the task represented by that node. For a particular state transition, the descriptor of the destination state is found by: (1) deleting the identity of the departing customer from the "T_i Customers" section of the descriptor of the current state; and (2) for each memory module j in the synchronization subsystem, adding one customer to the "O_j Customers" section if an operand packet destined for that memory module was generated. For a state with a non-empty "O_i Customers" section, whenever a chain O_i customer departs from the M/U service center, a transition takes place from the current state into a state identified by the following descriptor. Starting with the descriptor of the current state, we perform the following steps: (1) delete the identity of the departing customer from the "O_i Customers" section of the left part; and (2) update the incomplete operand sets listed in the $\{IOS_i\}$ section of the right part. Any operand set which becomes completed is immediately deleted from the $\{IOS_i\}$ section and a customer, corresponding to the task that became enabled, is added to the proper " T_j Customers" section of the state descriptor.

The initial state of the Markov process is described by listing those queueing network customers which represent the tasks enabled at the start of the program. We define a final state of the Markov process as a state which contains no active customers (i.e. the part of its descriptor to the left of ";" is empty), since no transition is possible out of such a state. To compute the steady state probabilities, all transitions to a final state are converted into transitions to the initial state. In fact, the corresponding Markov process represents a continuously repeated execution of the same program.

A detailed algorithm for generating a Markov model from a computation control graph, when modeling systems of the "Centralized Synchronization" categories, is presented in [KAP86].

4.2.2 State Space Reduction

In order to reduce the total number of states in the Markov model constructed, we can combine two or more "equivalent" states into a single, aggregate state. We define two states to be equivalent if they have identical transitions with identical rates to other states and if their respective descriptors have the same number of queueing network customers in each "T_i Customers" section and in each "O_i Customers" section. The descriptor of the aggregate state is formed by joining the individual descriptors of its constituents through the "/" symbol(s). That is, if "D1" and "D2" are the respective descriptors of two "equivalent" states, say S1 and S2, then the descriptor of the aggregate state is given by "D1/D2". Such state space reduction can significantly reduce the computational cost of solving the Markov process.

4.2.3 State Transition Rates

The departure rate of a chain T_i customer from the P/C service center, with n(T_j) customers of chain T_j present, for all possible j, is expressed as:

$$\mu_{T_i}\,(n(T_1)\,,\ldots,\,n(T_c)\,) \tag{4.1}$$

where c is the number of different customer chains for the P/C subnetwork. Due to the fact that all customers of the same chain have identical service demands and routing probabilities, the rate of a particular chain T_i customer departing is equal to:

$$\mu_{T_i}\,(n(T_1)\,,\ldots,\,n(T_c)\,)\,/\,n(T_i) \tag{4.2}$$

The departure rate of a particular chain O.i customer from the M/U service center is computed analogously and is expressed as:

$$\mu_{0_i}\,(n(0_1)\,,\ldots,\,n(0_m)\,)\,/\,n(0_i) \tag{4.3}$$

where m is the number of different customer chains in the M/U subnetwork (i.e., the number of distinct memory modules in the synchronization subsystem).

For a given state of a Markov process, the rate of leaving that state due to a departure of a particular chain T_i customer is computed from expression (4.2), with n(T_j) set to the number of customers listed in the "T_j Customers" section of that state's descriptor, for all j. The rate of making a particular state transition, associated with the departure of that customer, is equal to the probability of making that transition (i.e. the probability of enabling the corresponding feasible subset of arcs) multiplied by the rate given above. There is only one possible state transition associated with a departure of a particular chain O_i customer. Its rate is given by expression (4.3), with n(O_j) set to the number of customers listed in the "O_j Customers" section of that state's descriptor, for all j.

V. OPTIMIZATIONS AND HEURISTICS

Since the state space of a Markov process grows combinatorially with the number of nodes and arcs in a computation control graph, the solution procedure described in the previous section can be very expensive for large applications. In this section, we describe heuristic procedures for obtaining approximate solutions to some of the common programming constructs in a computationally efficient way. These procedures are based on successive aggregation of portions of a computation control graph (called segments). Each segment is described by its aggregate properties and a method of aggregating segments is developed. By successive aggregation of segements, one finally has the entire graph represented by a single segment with known performance measures.

Before proceeding, we need to define the notion of distribution of parallelism. For a particular physical system, supporting a total of c classes of tasks, the distribution of parallelism for a program segment S is given by the following discrete function of c-dimensional vectors of non-negative integers:

$$p_S\,(k_1\,,\ldots,\,k_c),\;\; k_i \textit{ is a non-negative integer, } i=1,\,\ldots,\,c \qquad (5.1)$$

The range of this function is the set of real numbers between 0 and 1 (inclusive) and the value of $p_S\,(k_1\,,\ldots,\,k_c)$ is the probability that (or fraction of time when), while segment S is being executed alone, there are exactly k_i enabled tasks of class i, i=1,...,c. Since, during every instant of a segment's execution time period, there is at least one enabled task, $p_S(0,\ldots,0) = 0$ for any S. If the computation control graph representing segment S is acyclic, then this is a finite distribution. Note that the same program segment will, in general, have different distributions of parallelism with different system architectures.

5.1 Segments: Compact Description

Each program segment can be solved as a stand-alone program, using the procedure developed in the previous section, to obtain its mean execution time, average number of tasks executed, and distribution of parallelism. Thus, for a given segment S, we can "ignore" its underlying computation control graph and describe it by the following triple:

$$\{N_S, T_S, p_S(k_1, \ldots, k_c)\}, \tag{5.2}$$

where T_S is its mean execution time, $p_S(k_1, \ldots, k_c)$ is its distribution of parallelism, and c is the number of <u>all</u> task classes supported by the system. Note that the tasks of an individual segment may only represent a subset of those classes. If c=1, then N_S is the average number of tasks executed during an invocation of segment S. If c>1, then N_S is the vector $(N_{S,1}, \ldots, N_{S,c})$, where $N_{S,i}$ is the average number of tasks of class i executed during an invocation of S. This compact description captures only the "steady-state" properties of a segment and ignores its time-dependent behavior. This is analogous to making "fluid flow" approximations when modeling queueing systems [KLE76].

5.2 Sequential Combination

First we will consider a <u>sequential</u> combination, S, of two segments, segment A and segment B. The mean execution time of this combination, $\underline{T_S}$, is equal to $T_A + T_B$.

$$p_S(k_1, \ldots, k_c) = \frac{[p_A(k_1, \ldots, k_c) \cdot T_A + p_B(k_1, \ldots, k_c) \cdot T_B]}{T_S} \tag{5.3}$$

The average number of tasks executed during an invocaton of S is $N_S = N_A + N_B$.

A sequential combination of several segments can be solved in an analogous fashion.

5.3 EXCLUSIVE-OR Combination

Next we will study an EXCLUSIVE-OR combination, E, of segments $S_1, S_2, \ldots, S_n$, which is shown in Figure 5.1. In this combination, exactly one out of n possible segments is executed during each invocation of E, with segment S_j being executed with probability w_j, j=1,...,n (the sum of w_j's must equal to one). In other words, $100 \cdot w_j$ percent of the invocations of the combination E will "behave" like segment S_j, j=1,...,n.

The mean execution time of E is given by:

$$T_E = \sum_{j=1}^{n} w_j \cdot T_{S_j} \qquad (5.4)$$

Thus, the fraction of time when combination E will "behave" like segment S_j is $w_j \cdot T_{S_j}/T_E$, j=1,...,n. The distribution of parallelism for the combination is given by:

$$p_E(k_1, \ldots, k_c) = \frac{\sum_{j=1}^{n} w_j \cdot T_{S_j} \cdot p_{S_j}(k_1, \ldots, k_c)}{T_E} \qquad (5.5)$$

The average number of tasks executed during an invocaton of E is:

$$N_E = \sum_{j=1}^{n} w_j \cdot N_{S_j} \qquad (5.6)$$

5.4 Parallel Combination

Now we will analyze a parallel combination, P, of segments A and B. The following development is applicable to solving models with only one class of tasks (i.e. c=1). (With c>1, some of the "independence" and "fluid flow" assumptions necessary for this development are no longer valid.)

5.4.1 Computing T_P

Let $T_{A|B}$ be the mean time to execute segment A if segment B started to execute at exactly the same time that segment A did. $T_{B|A}$ is defined analogously. Thus, T_P is equal to either $T_{A|B}$ or $T_{B|A}$, depending on which segment completes last. Without loss of generality, let us suppose that segment A is first to finish and let $\hat{B}$ be the portion (i.e., the fraction of total number of tasks) of segment B which was completed during the time that A had been executing (i.e., during the first $T_{A|B}$ time units of $T_{B|A}$).

$\hat{B}$ may contain a nonintegral number of tasks if some tasks were only partially completed. The time to finish executing segment B is given by $T_B - T_{\hat{B}}$. ($T_{\hat{B}}$ is the average time it would have taken to execute the $\hat{B}$ portion of segment B if B were running alone.) Thus,

$$T_P = T_{B|A} = T_{A|B} + (T_B - T_{\hat{B}}) \tag{5.7}$$

In the case that segment B completes first,

$$T_P = T_{A|B} = T_{B|A} + (T_A - T_{\hat{A}}) \tag{5.8}$$

with segment $\hat{A}$ analogously defined.

We will now show how to determine which segment completes first (on the average) and how to estimate $T_{A|B}$, $T_{B|A}$, $T_{\hat{A}}$, and $T_{\hat{B}}$.

5.4.1.1 Notation

N_A and N_B are the average numbers of tasks executed during invocations of segments A and B, respectively. $\mu(k)$ is the throughput (or mean departure rate) of tasks of the distributed system being modeled, when k tasks are being executed concurrently, k=1,2,... Let μ_A and μ_B be the <u>average</u> throughputs of tasks of segments A and B, respectively, when each is being executed <u>alone:</u>

$$\mu_A = N_A / T_A \tag{5.9}$$

$$\mu_B = N_B / T_B \tag{5.10}$$

Let μ_A and μ_B be the average task throughputs of segments A and B, respectively, during the time period when both segments are executing <u>concurrently.</u>

5.4.1.2 Estimating μ'_A and μ'_B

The formula for computing μ'_A is given by:

$$\mu'_A = \mu\,(i+j) \cdot \frac{i}{(i+j)}\, Pr\,(i \text{ of } A, j \text{ of } B \mid A\&B) \tag{5.11}$$

Each component in the summation is a product of two terms. The first term, $\mu(i+j) \cdot i/(i+j)$, is the throughput of segment A tasks, when there are i segment A tasks and j segment B tasks "competing" for system resources. This term is derived from the fact that the P/C subnetwork in the original physical domain model is approximated by a single state-dependent service center which adheres to the Processor-Sharing scheduling discipline. (Note that this expression holds true exactly if and only if the P/C subnetwork is equivalent to a single processor-sharing resource.) The second term represents the joint probability of exactly i tasks of segment A and exactly j tasks of segment B being enabled, while both segments are executing in parallel. In order to obtain the exact value of this term, one must construct and solve the complete Markov process for combination P, which defeats the purpose of our approximation procedure. Thus, we will attempt to estimate this joint probability using only already known values.

First, observe that:

$$Pr\,(i \text{ of } A \text{ and } j \text{ of } B \mid A\&B) = Pr(i \text{ of } A \mid A\&B) \cdot Pr(j \text{ of } B \mid i \text{ of } A \mid A\&B) \tag{5.12}$$

Now, we will approximate Pr(j of B | i of A | A&BA) by Pr(j of B | A&B). This approximation is motivated by the observation that the precedence relationships of tasks in one segment are independent of those in the other segment. However, note that Pr(i of A | A&B) and Pr(j of B | A&B) are not truly independent of each other.

Next, we will replace Pr(i of A | A&B) by p_A(i) and Pr(j of B | A&B) by p_B(j). This approximation is based on the assumption that the relative throughputs of segment A tasks are not dependent on the number of segment B tasks present, and vice-versa. The more linear the μ(k) function is, the more accurate the latter estimates are.

Finally, combining all of the equations presented above, we get:

$$\mu'_A = \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \mu\,(i+j) \bullet \frac{i}{(i+j)}\, p_A\,(i) \bullet p_B\,(j) \tag{5.13}$$

The formula for estimating μ'_B can be derived in an analogous fashion and is presented below.

$$\mu'_B \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \mu\,(i+j) \bullet \frac{j}{(i+j)}\, p_A\,(i) \bullet p_B\,(j) \tag{5.14}$$

5.4.1.3 Estimating $T_{\hat{A}}$ and $T_{\hat{B}}$

First, let us consider the case where segment A completes execution before segment B does. $T_{\hat{A}}$ is then simply equal to T_A. The average number of segment B tasks completed during the execution of segment A, $N_{\hat{B}}$, is given by $\mu'_B \bullet T_{A|B}$. If segment B were executing alone, its average task throughput rate would be μ_B. Thus, $T_{\hat{B}}$ can be approximated by:

$$T_{\hat{B}} = \frac{N_{\hat{B}}}{\mu_B} = \frac{(\mu'_B \bullet T_{A|B})}{\mu_B} \tag{5.15}$$

If segment B is first to finish, then:

$$T_{\hat{A}} = \frac{N_{\hat{A}}}{\mu_A} = \frac{(\mu'_A \bullet T_{B|A})}{\mu_A} \tag{5.16}$$

5.4.1.4 Computing $T_{A|B}$ and $T_{B|A}$

Our first step is to determine which one of the two segments completes execution first (on the average). Toward this goal, we prove the following lemma.

Lemma 5.1: Under the assumptions and definitions stated above,

$$N_A/\mu'_A < N_B/\mu'_B \text{ if and } \textit{only} \text{ if } T_{A|B} < T_{B|A}$$

Proof: Suppose that $N_A/\mu'_A < N_B/\mu'_B$ but $T_{A|B} > T_{B|A}$. Then the execution of segments A and B is represented by the timing diagram shown in Figure 5.2. From this diagram, we obtain the following relationship:

$$\mu'_A \bullet T_{B|A} < N_A \rightarrow T_{B|A} < N_A/\mu'_A \rightarrow N_B/\mu'_B < N_A/\mu'_A$$

Thus, we obtain a contradiction.

Conversely, we can analogously show that, assuming that $T_{A|B} < T_{B|A}$ while $N_A/\mu'_A > N_B/\mu'_B$, also leads to a contradiction.

Q.E.D.

Thus, in order to compute $T_{A|B}$ and $T_{B|A}$, we need only to obtain μ'_A and μ'_B and apply Lemma 5.1. If $N_A/\mu'_A < N_B/\mu'_B$, then segment A completes execution first, $T_{A|B} = N_A/\mu'_A$, and $T_{B|A} = T_{A|B} + (T_B - \hat{T_B})$. If $N_A/\mu'_A > N_B/\mu'_B$, then segment B completes execution first, $T_{B|A} = N_B/\mu'_B$, and $T_{A|B} = T_{B|A} + (T_A - \hat{T_A})$.

5.4.2 Computing p_P(k)

Without loss of generality, we will assume that segment A is first (on the average) to complete execution. Let s(k), k≥0, be the distribution of parallelism of combination P during the first $T_{A|B}$ time units of the execution time period of P, i.e. s(k) is the "joint" distribution of parallelism of segments A and B. Let r(k), k≥0, be the distribution of parallelism of P during the remainder of the combination's execution time. Applying arguments similar to those used to derive equations (5.13) and (5.14), s(k), for all values of k, can be estimated by:

$$s(k) = \sum_{i=0}^{k} p_A(i) \cdot p_B(k-i) \tag{5.17}$$

Using the "fluid flow" assumption that the distribution of parallelism of segment B remains uniform over all subintervals of T_B, r(k) is approximately equal to p_B(k). Finally, from Bayesian principles, it follows that, in the case that segment A finishes first, p_P(k) can be estimated by:

$$p_P(k) = \frac{[s(k) \cdot T_{A|B} + r(k) \cdot (T_p - T_{A|B})]}{T_p} \text{ for } \textit{all } k \tag{5.18}$$

From the symmetry of the preceding derivation, in the case that segment B finishes first,

$$p_P(k) = \frac{[s(k) \cdot T_{B|A} + r(k) \cdot (T_P - T_{B|A})]}{T_P} \text{ for } \textit{all } k \tag{5.19}$$

with r(k) now being approximated by p_A(k).

5.4.3 Parallel Combination of Several Segments

A parallel combination of several segments can be solved by an iterative application of the procedures presented above. Suppose that there are n segments to begin with. We can take two of those segments, solve them as a parallel combination, replace them by a single, aggregate segment, and end up with a total of n-1 segments. We now proceed in this fashion until only one segment is left, which is then the desired solution. As can be seen from the formulas given above, the algorithm for combining segments in parallel possesses both commutative and associative properties, which allows us to combine segments in any order.

5.5 Parallel Combination of Sequential Combinations

The type of segment combination we are considering here is illustrated in Figure 5.3. A is a sequential combination of segments A_1, A_2, A_3 ,...., A_n. B is a sequential combination of segments B_1, B_2,..., B_m. The result of our solution process will be a sequential combination P, consisting of segments P_1, P_2 ,...., P_q, where max(n,m) ≤ q ≤ n+m. The exact value of q can only be determined at the end of the solution process.

Our first step is to solve a parallel combination of segments A_1 and B_1, in order to obtain segment P_1. Without loss of generality, let us suppose that segment A_1 completes execution before segment B_1 does. Then P_1 will be defined by the following segment descriptor: $\{N_{P_1}, T_{P_1}, p_{P_1}(k)\}$, where $N_{P_1} = N_{A_1} + N_{\hat{B}_1}$ and $T_{P_1} = T_{A_1|B_1}$. $p_{P_1}(k)$ is the joint distribution of parallelism of segments A_1 and B_1, and can be estimated in the same fashion as s(k) in equation (5.17). P can now be redefined as the sequential combination of segment P_1 and of the result of solving the combination P', as depicted in Figure 5.4(a). In this combination, segment $\tilde{B}_1$ is the complement of segment $\hat{B}_1$, with respect to segment B_1.

In the case that segment B_1 completes execution first, P_1 will be analogously defined, with $N_{P_1} = N_{B_1} + N_{\hat{A}_1}$ and $T_{P_1} = T_{B_1|A_1}$. The portion of the original combination remaining after this step, P', is shown in Figure 5.4(b), with segment $\tilde{A}_1$ being the complement of segment $\hat{A}_1$, with respect to segment A_1.

The next step is to solve a parallel combination of segments A_2 and $\tilde{B}_1$ (or, depending on the result of the first step, $\tilde{A}_1$ and B_2) to obtain segment P_2. Thus, by iteratively "reducing" the original combination, we can proceed to solve for segments P_2, P_3, P_4 ,.... until we "run out" of either A-segments or B-segments. The "leftover" segments can now be simply "appended", in a sequential order, to the portion of P that has been obtained so far, in order to produce the final solution. At each step, we "eliminate" either one A-segment or one B-segment, or, in the case that the segments (being then considered) complete execution simultaneously, both an A-segment and a B-segment. Therefore, the maximum number of steps involved in solving this combination is n+m.

A parallel combination of several sequential combinations can be solved in a way analogous to solving a parallel combination of several single segments.

5.6 Other Heuristics

We have developed similar approximation procedures for solving models of various looping constructs, ranging from the simple loop of Figure 5.5(a) to the nested loop shown in Figure 5.5(b), which are beyond the scope of this article. It is important to note that those techniques, as well as the ones described in detail above, can all be applied hierarchically when analyzing complex program structures. Furthermore, we have also generated a heuristic approach for analyzing random arrivals of programs or processes.

Details of the techniques referred to in the preceding paragraph can be found in [KAP86].

VI. APPLICATION EXAMPLE

In this section, we give an example of an application of our modeling methodology.

6.1 Program Description

The particular process, P, we have chosen is the one considered in the performance modeling study by Chu and Leung [CHU84]. The computation control graph for this process is shown in Figure 6.1. It has a well structured form and can be conveniently decomposed into a hierarchical combination of segments. This hierarchical decomposition is illustrated in Figure 6.2, where process P is represented at seven different levels of abstraction. On level 0, segment P represents the whole process. On level 1, segment P is revealed to be the sequential combination of task 1, segment A, and task 15.

On the second level, segment A is detailed as a simple loop, having segment B as its body. The third level allows us to view segment B as the sequential combination of task 2, segment C, and task 14. Level 4 reveals that segment C is the EXCLUSIVE-OR combination of segments D, E, and F. On the fifth level, we find that: segment D is the sequential combination of tasks 3 and 12; segment E is the sequential combination of tasks 4, 6, and 9; and segment F is the sequential combination of task 5, segment G, and task 13. Finally, on level 6, segment G is represented by the computation control graph shown in Figure 6.2(g).

The computation of segment G's segment descriptor involves constructing and solving the Markov process depicted in Figure 6.3. Descriptors of segments D, E, and F are found by applying the sequential combination algorithms. Segment C is solved using the EXCLUSIVE-OR combination techniques. Proceeding in the same fashion, we can compute the descriptors of segments B, A, and, finally, P itself. Since only sequential and EXCLUSIVE-OR segment aggregation techniques are employed, the solution approach described above is applicable even if tasks of process P belong to different classes.

6.2 System Description

The architecture of the distributed system we are evaluating is shown in Figure 6.4. It is based on the Cm* multiprocessor [SWA77]. The basic component of this architecture is a Computer Module. Four such components are present in our system, being interconnected through a communication bus. Each Computer Module consists of a processor, an intelligent switch, and a storage unit with the associated controller. All memory references made by a processor are sent to and interpreted by the switch of that Computer Module. All local (with respect to the Computer Module) requests are forwarded directly to the attached storage unit controller. Whenever a remote memory request is issued, it is intercepted by the local switch, converted into a request to the switch of the proper Computer Module, and transmitted as a data packet over the communication bus. Upon receiving this request, the remote switch converts it into the proper memory reference to its local storage. If data is to be returned to the requesting switch, it is packetized and sent back via the bus. Remote memory requests are also used by processors to exchange messages among themselves.

In this configuration, Computer Module 0 is designated to perform the synchronization of tasks, while modules 1, 2, and 3 are dedicated to executing enabled tasks. All of the incomplete operand sets are maintained in the local memory of Computer Module 0. When an operand packet is received, it is first stored in memory by the switch and then the processor is notified, which updates the affected operand sets. The tasks corresponding to the operand sets which were completed are enabled and each is sent to either module 1, 2, or 3 for execution, the particular module being chosen at random. When Computer Module i, i=1,2,3, receives a task from Computer Module 0, it first acquires the necessary data from its local storage unit, then executes the task code, and, finally, generates a result packet (which also serves as an operand packet) and returns it to Computer Module 0.

6.3 Physical Domain Model

The system described above belongs to the "Dynamic Allocation with Centralized Synchronization" category. Its physical domain model is depicted in Figure 6.5. It consists of four "clusters" of service centers, CM0, CM1, CM2, CM3, and of service center BUS. Cluster CMi, i=0,1,2,3, which represents the resources of Computer Module i, consists of service centers Pi, Mi, and Si. P0 models the time needed to process a received operand packet. M0 represents storage access delays incurred in updating incomplete operand sets. Service center S0 models processing of local memory references, receiving of operand packets from the communications bus, and forwarding enabled tasks to the bus for transmission to other computer modules. Together, service centers P0, M0, and S0 represent the task synchronization overhead of our system. Pi, i=1,2,3, models the execution and waiting times of a task at Computer Module i. Delays at server Mi, i=1,2,3, correspond to fetching the data, which is needed to execute a received task, from the local storage unit of module i. Si represents receiving of tasks (sent to module i by module 0) from the communications bus, processing of storage access requests, and transfering of result packets (generated by completed tasks) to the bus for transmission to Computer Module 0. Service center BUS represents the contention for the communications bus and the transmission time of tasks and result packets.

Each queueing network customer in this P/C subnetwork represents either a task, a result packet, an operand packet, or a storage access request, depending on the stage of its "lifetime". A customer first visits service centers S0 and M0, representing a storage access request. It then proceeds to S0 and P0, while modeling an operand packet. Afterwards, representing the transmission of an enabled task to Computer Module i (i is chosen randomly as either 1, 2, or 3) it again visits server S0, followed by BUS and Si. As a local memory reference, it proceeds to Mi and then back to Si. During the subsequent visit to service center Pi, it models the task submitted for execution. Finally, representing a result packet, it returns to Si and then completes its route at BUS service center. Different customer classes are used in this queueing network in order for customers to be able to make proper routing decisions.

6.4 Numerical Results

The timings that were assumed in this example for various operations are listed in Table 6.1. All service times are given as means of the exponential distribution and do not include any associated queueing delays. Using these values, the average time needed to execute a single instance of process P was estimated by our methodology to be 33.64. The result given by a detailed simulation of this environment was 34.18. Thus, our estimate was within 1.6% of the "actual" value.

Space limitation precludes the presentation of other studies that have been carried out to test the accuracy of the approach. In general, we have found that the approximate analysis yields results that are within 10% of the corresponding simulation results.

VII. CONCLUSIONS

We believe that our methodology can be applied to evaluating the performance of distributed architectures at any level of the computer systems hierarchy. Its applicability is not limited to specific structures of programs nor to certain types of physical systems. The same, general modeling principles can be utilized in each individual case. When representing the system architecture of a particular execution environment (by a physical domain model), a modeler does not have to be concerned about what specific programs are actually going to be evaluated with that system. Only the number of different types of tasks and the attributes of each type have to be known. Conversely, when modeling the execution of a specific program, only the category of the system where the program is to be run has to be known in order to be able to generate the structure (not the transition rates) of the corresponding Markov process. Such "loose coupling" between models of program behavior and models of system architectures is especially beneficial during system design and early development stages, when the intended applications are not yet known.

It is important to emphasize that the solutions obtained by applying the analytic techniques presented in this paper are estimates of the actual values and should be treated as such. The quality of approximation will vary with each particular case. However, we do believe that, in most cases, the results yielded by our methodology will provide a reasonable indication of the relative performance of the system being considered, with respect to competing architectures or different parameter selections. Thus, we recommend that the application of this technique be focused on "exploratory" design studies, followed by a simulation of each selected candidate.

With respect to general-purpose simulation packages, our methodology requires substantially less processing time, in both model development and solution phases. In virtually all cases, the design and implementation of a model using a general-purpose simulation language would be significantly more time consuming than the construction of the corresponding Markov process. More importantly, the computation of state transition rates and the solution of the Markov process would, in general, take several orders of magnitude less time than the required simulation runs.

With respect to other currently available analytic methods, the following observations can be made. Techniques based only on queueing networks do not support explicit representation of interdependencies of tasks and, thus, they are harder to use and, in general, will yield less accurate estimates than our techniques. Furthermore, such methods are usually applicable to modeling only very simple precedence relationships, e.g., fork-join constructs, and cannot capture the behavior of complex program structures. As far as graph-based methods go, there are two basic categories. The methods of the first type assume either that the capacity of each system resource is infinite or some other overly simplified archi tecture. Those methods, although usually reasonably accurate, are very limited in their scope of practical application, since they are not capable of modeling (much more complex) features of realistic systems, which can be easily handled by our methodology. In the second category, all architectural details included in a model are represented by nodes and arcs in a graph, intermixed together with the representation of a program's precedence relationships. Such techniques, e.g., Stochastic Petri Nets [MOL81], are prone to rapid state space explosion as the number of system components increases. (In our methodology, the complexity of a Markov process is not dependent on the system size; the cost of computing state transition rates increases with the system size, but <u>not</u> nearly as fast!) Also, a minor architectural change may require that a completely new graph be constructed and solved. Thus, compared to our methodology, the latter graph models are much more "expensive" to both generate and solve and they are not work conserving.

REFERENCES

[ACK82]	Ackerman, W. B., "Data Flow Languages", *Computer, 15(2),* February 1982, pp. 15-25.
[CHU84]	Chu, W. W. & K. K. Leung, ''Task Response Time Model and Its Applications for Real-Time Distributed Processing Systems,'' *Proceedings,* Real-Time Systems Symposium, December 1984.
[COU77]	Courtois, P. J., *Decomposability, Queueing and Computer System Application,* Academic Press, New York, 1977.
[DEN72]	Dennis, J. B., J. B. Fosseen & J. P. Linderman, ''Data Flow Schemas,'' Research Report, MIT, July 1972.
[EST78]	Estrin G., ''A Methodology for Design of Digital Systems -- Supported by SARA at the Age of One,'' *AFIPS Proceedings,* National Computer Conference, 1978.
[FER72]	Fernandez, E., ''Activity Transformations on Graph Models of Parallel Computations,'' Ph.D. Thesis, Computer Science Dept., UCLA, Los Angeles, CA, October 1972.
[HEI83]	Heidelberger, P. & K. S. Trivedi, ''Analytic Queueing Models for Programs with Internal Concurrency,'' *IEEE Trans. on Computers,* Vol. C-32, No. 1, January 1983, pp. 73-82.
[INF81]	Information Research Associates, ''PAWS -- Performance Analyst's Workbench System: Introduction and Technical Summary,'' I.R.A., Austin, TX, 1981.
[KAP86]	Kapelnikov, A., ''Analytic Modeling Methodology for Evaluating the Performance of Distributed Multiple Computer Systems'', Ph.D. Dissertation, UCLA Computer Science Department, December 1986.
[KAR67]	Karp, R. M., R. E. Miller & S. Winograd, ''The Organization of Computations for Uniform Recurrence Equations,'' *JACM,* Vol. 14, 1967, pp. 563-590.
[KUC77]	Kuck, D. J., "A Survey of Parallel Machine Organization and Programming", *ACM Comp. Surveys, Vol. 9, No. 1,* March 1977, pp. 29-59.
[LAV82]	Lavenberg, S. S., E. A. MacNair, H. M. Markowitz, C. H. Sauer, G. S. Shedler & P. D. Welch, *Computer Performance Modeling Handbook,* Academic Press, 1982.
[MOL81]	Molloy, M., ''On the Integration of Delay and Throughput Measures in Distributed Processing Models,'' Ph.D. Dissertation, UCLA, September 1981.
[PET81]	Peterson, J. L., *Petri Net Theory and the Modeling of Systems,* Prentice Hall, New Jersey, 1981.

[SAU81b] Sauer, C. H. & E. A. MacNair, *Computer/Communication System Modeling with the Research Queueing Package, Version 2*, IBM T. J. Watson Research Center, Yorktown Heights, New York, May, 1981.

[SWA77] Swan, R. J., et al., "The Implementation of the Cm* Multiprocessor", *Proceedings,* National Computer Conference, 1977.

[THO81] Thomasian, A. and P. Bay, "Performance Analysis of Task Systems Using a Queueing Network Model", *Proceedings,* International Workshop on Timed Petri Nets, Torino, Italy, July 1985.

[VAN78] Vantilborgh, H., "Exact Aggregation in Exponential Queueing Networks", *JACM, Vol. 25,* 1978, pp. 620-629.

[VER82] Vernon, M. K., ''Performance-Oriented Design of Distributed Systems,'' Ph.D. Dissertation, UCLA, December 1982.

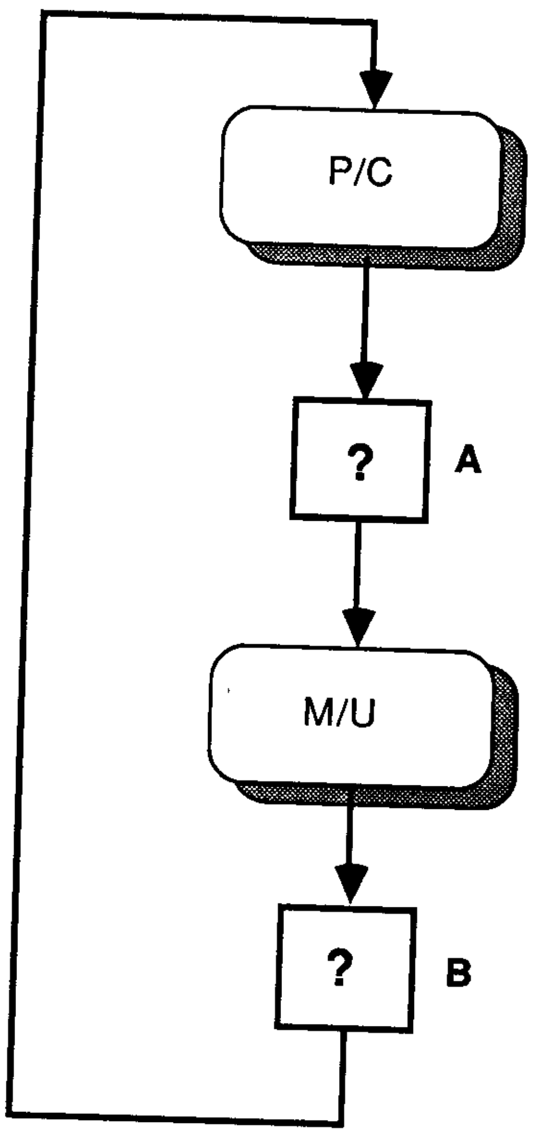


Figure 3.1 Physical Domain Model

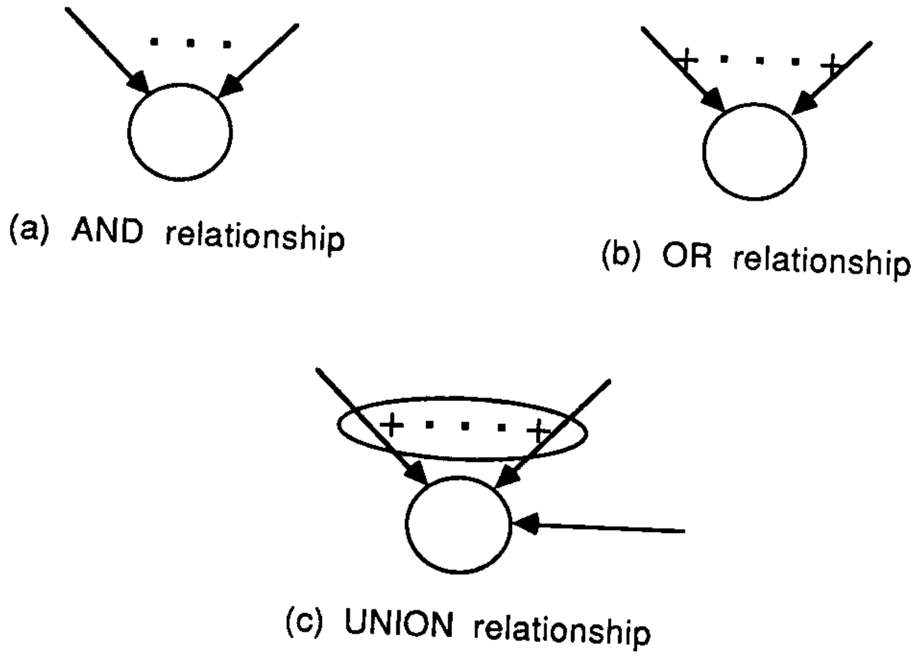
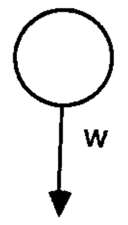
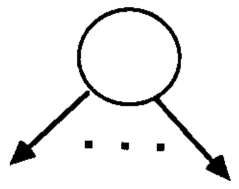


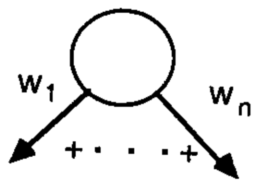
Figure 3.2 Relationships Between Arcs Terminating at the Same Node



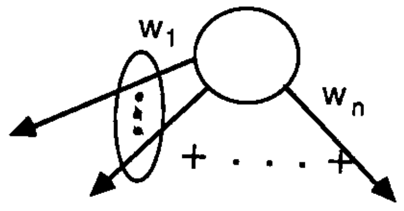
(a) Probability weights



(b) AND relationship



(c) EXCLUSIVE-OR relationship



(d) UNION relationship

Figure 3.3 Relationships Between Arcs
Emanating from the Same Node

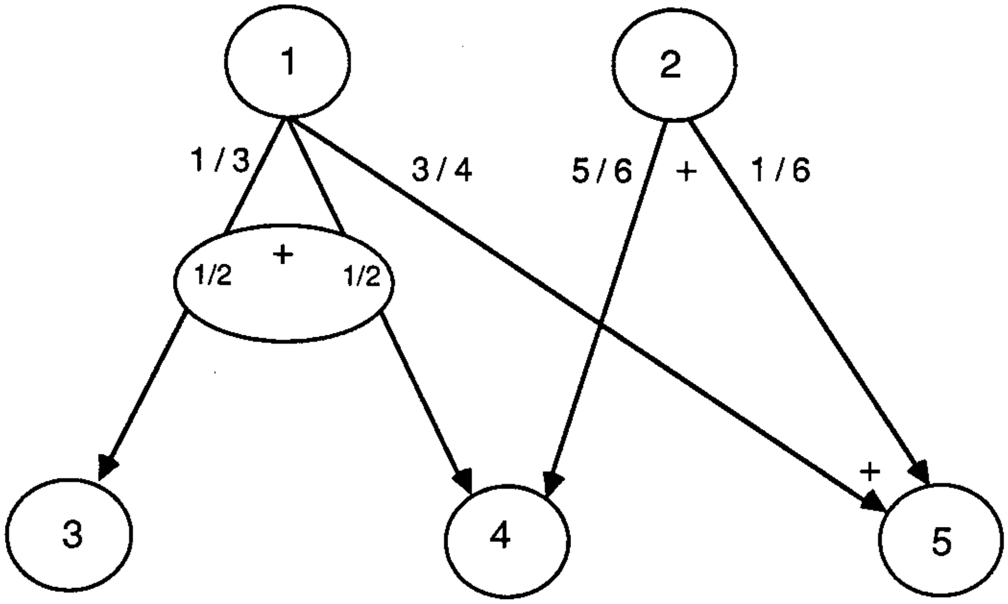
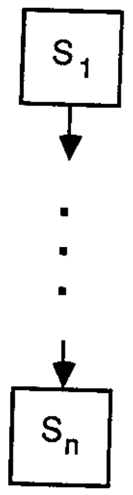
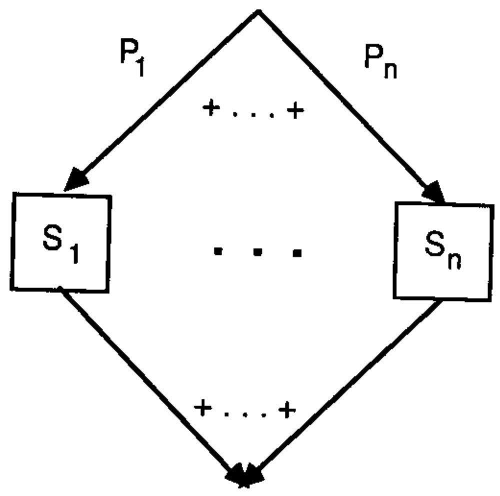


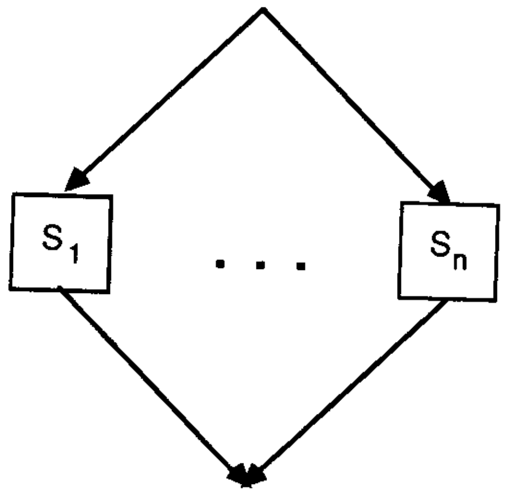
Figure 3.4 Example of a Computation Control Graph



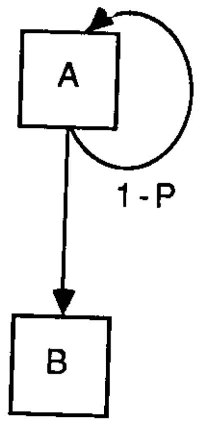
(a) Sequential Construct



(b) Exclusive-OR Construct



(c) Parallel Construct



(d) Loop Construct

Figure 3.5 Common Programming Constructs

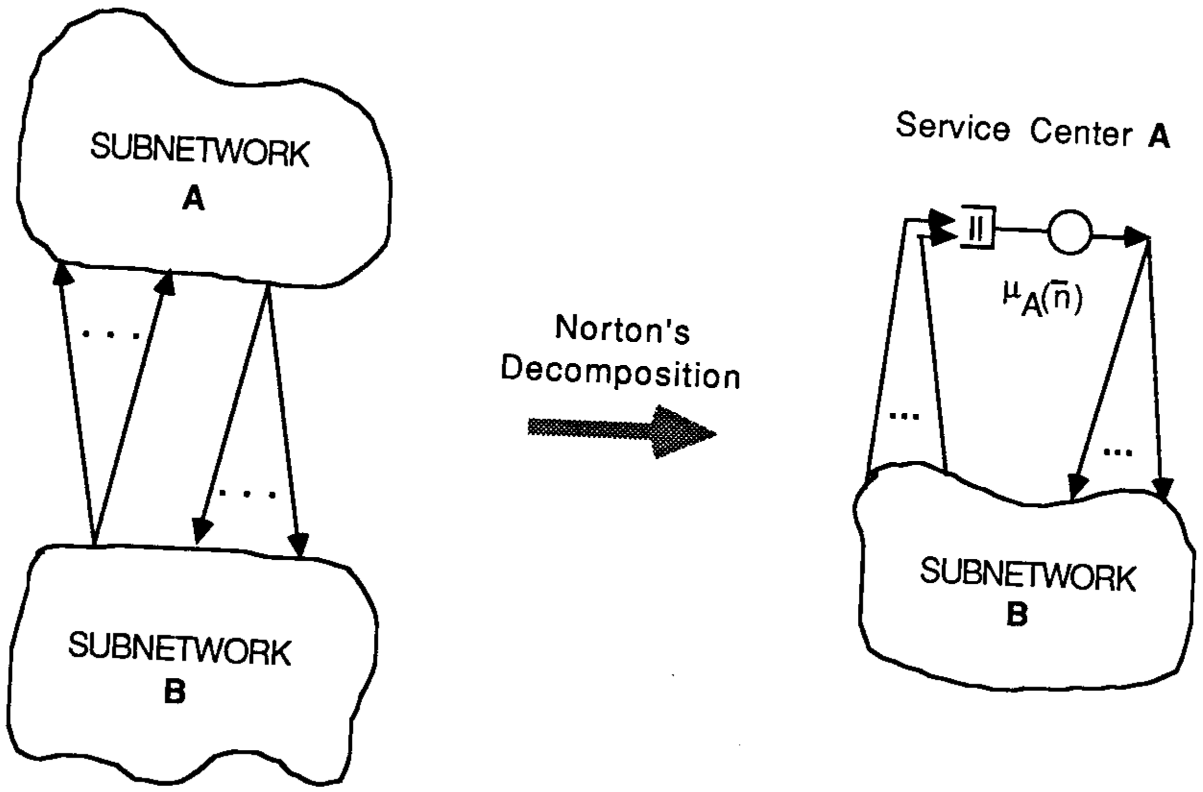


Figure 4.1 Norton's Decomposition of a Queueing Network

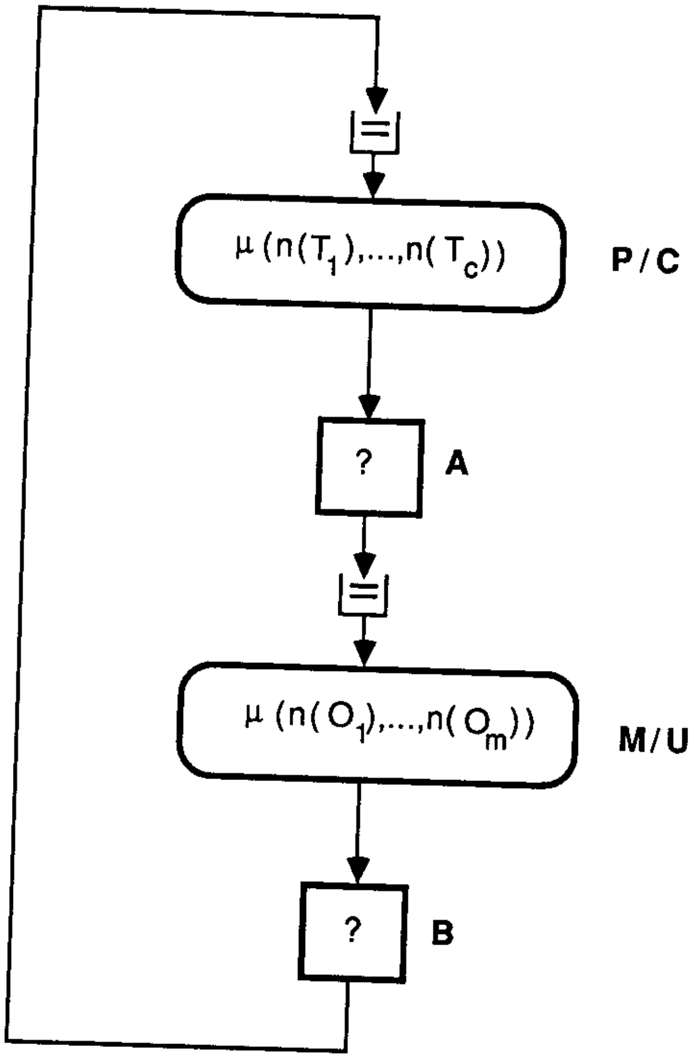


Figure 4.2 Physical Domain Model after Applying Decomposition Approximation

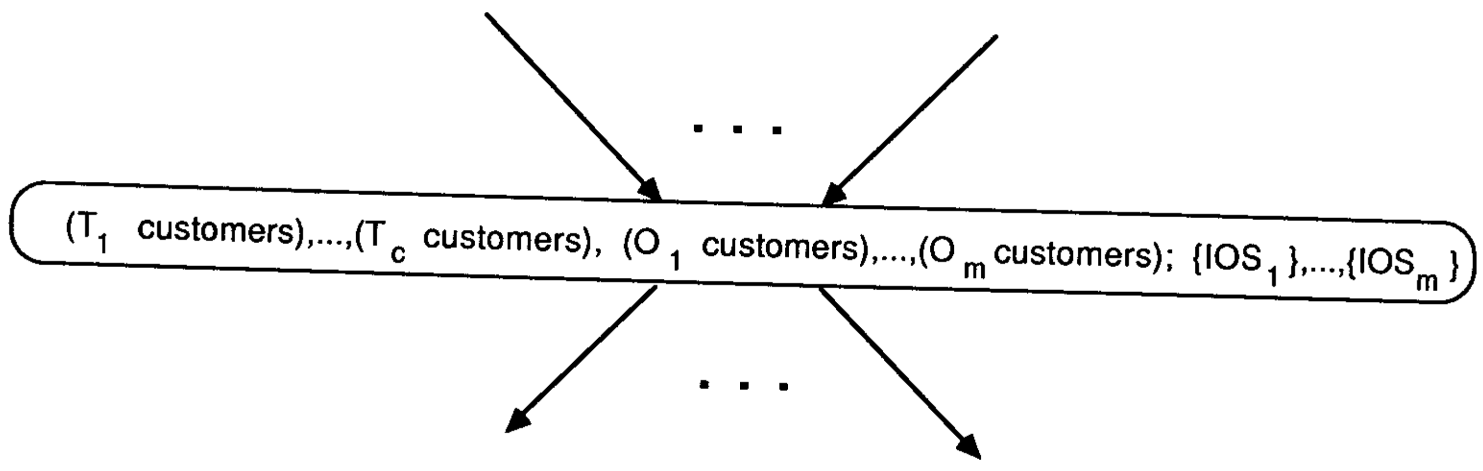

$(T_1$ customers$),\ldots,(T_c$ customers$)$, $(O_1$ customers$),\ldots,(O_m$ customers$)$; $\{IOS_1\},\ldots,\{IOS_m\}$
...
...

Figure 4.3 Descriptor of a State in a Markov Process

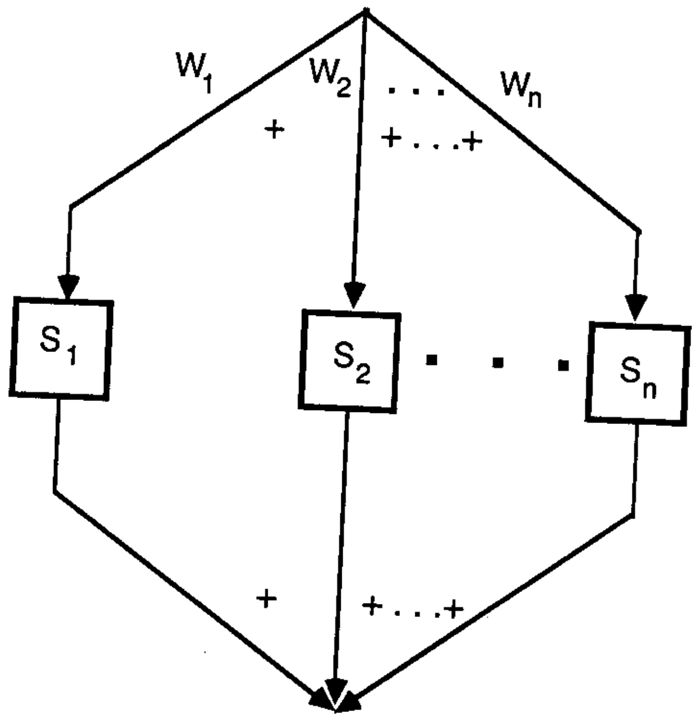


Figure 5.1 Exclusive-OR Combination

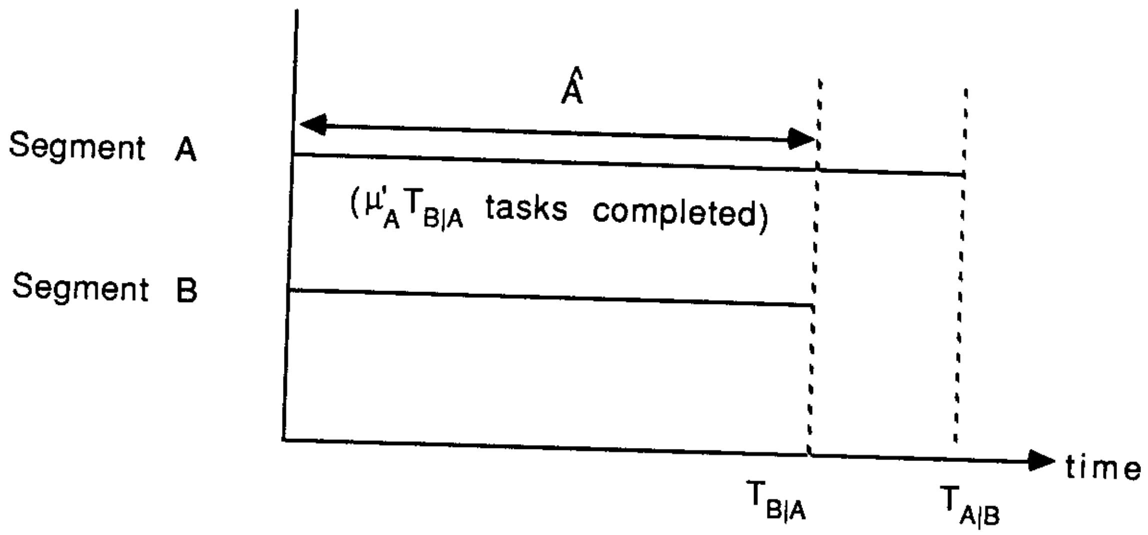


Figure 5.2 Parallel Execution of Two Segments

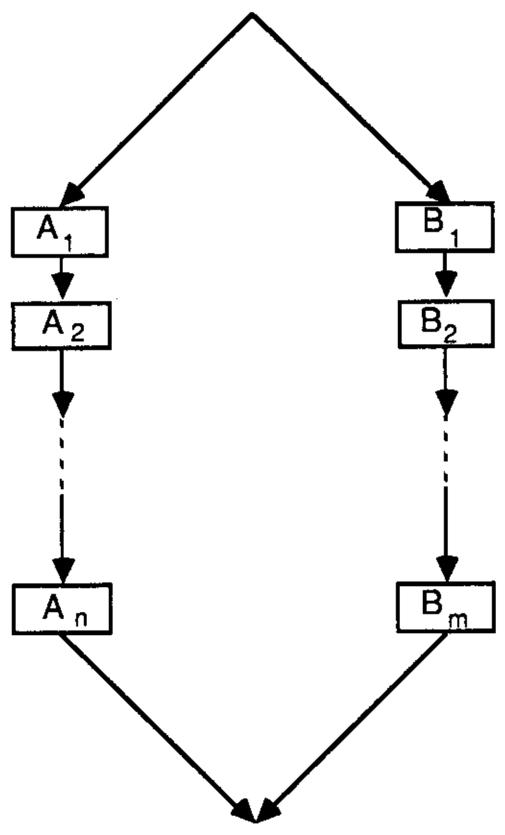
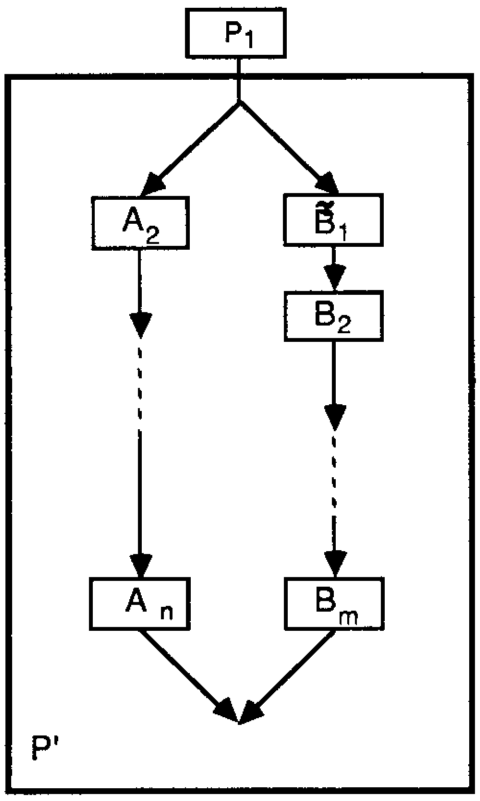
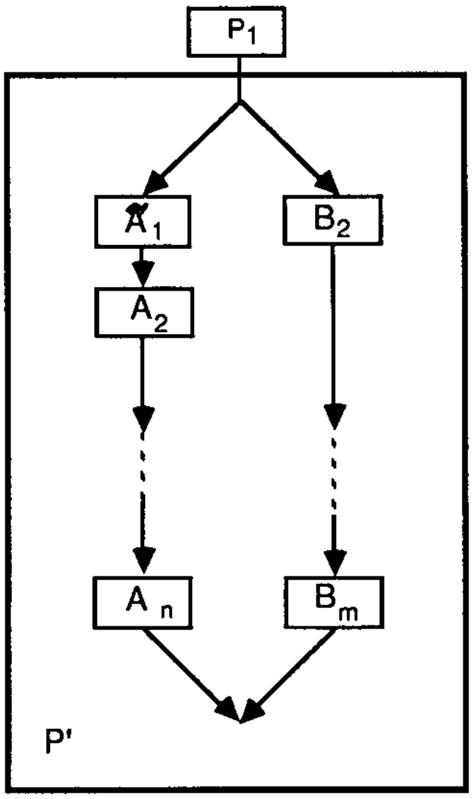


Figure 5.3 A Parallel Combination of Sequential Combinations

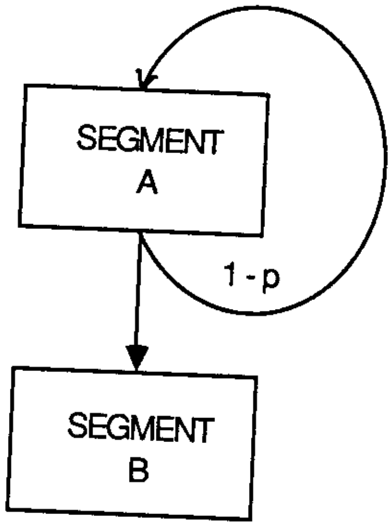


(a) A_1 is first to finish

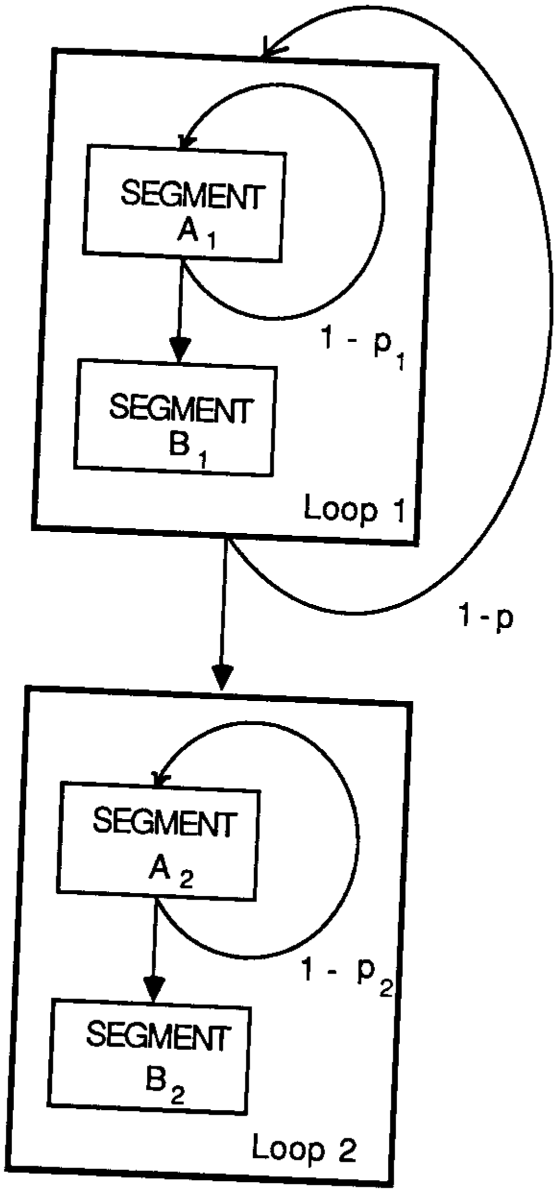


(b) B_1 is first to finish

Figure 5.4 Result of Completing the First Step in Solving for P



(a) Simple loop



(b) Nested loop

Figure 5.5 Examples of Looping Constructs

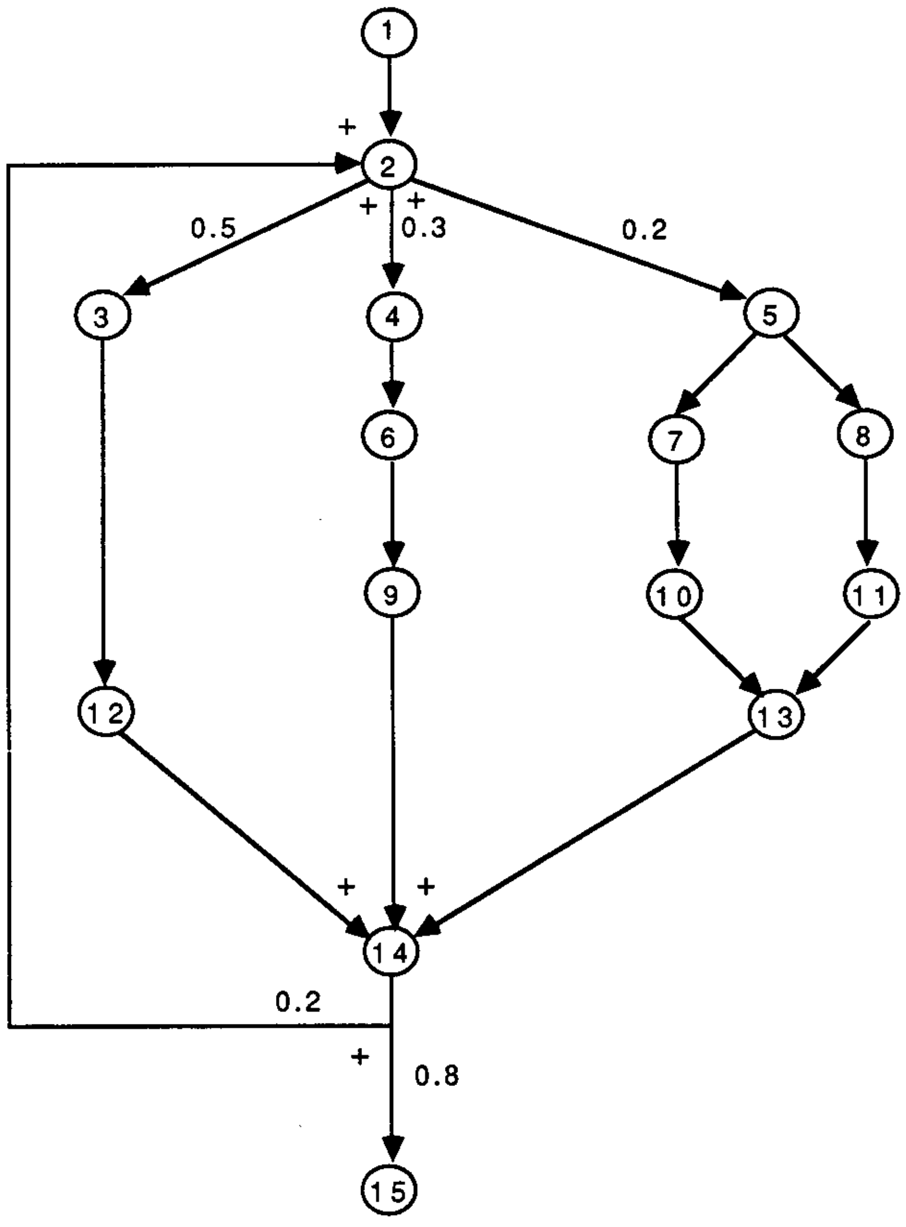


Figure 6.1 Computation Control Graph for Process P

Table 6.1 Timings for the Complex System

Operation	Average Time (s)
EXECUTION OF A TASK	2
STORAGE ACCESS	0.5
MATCHING OF OPERANDS	0.2
TASK TRANSMISSION	0.2
RESULT PACKET TRANSMISSION	0.2
SWITCH PROCESSING	0.1

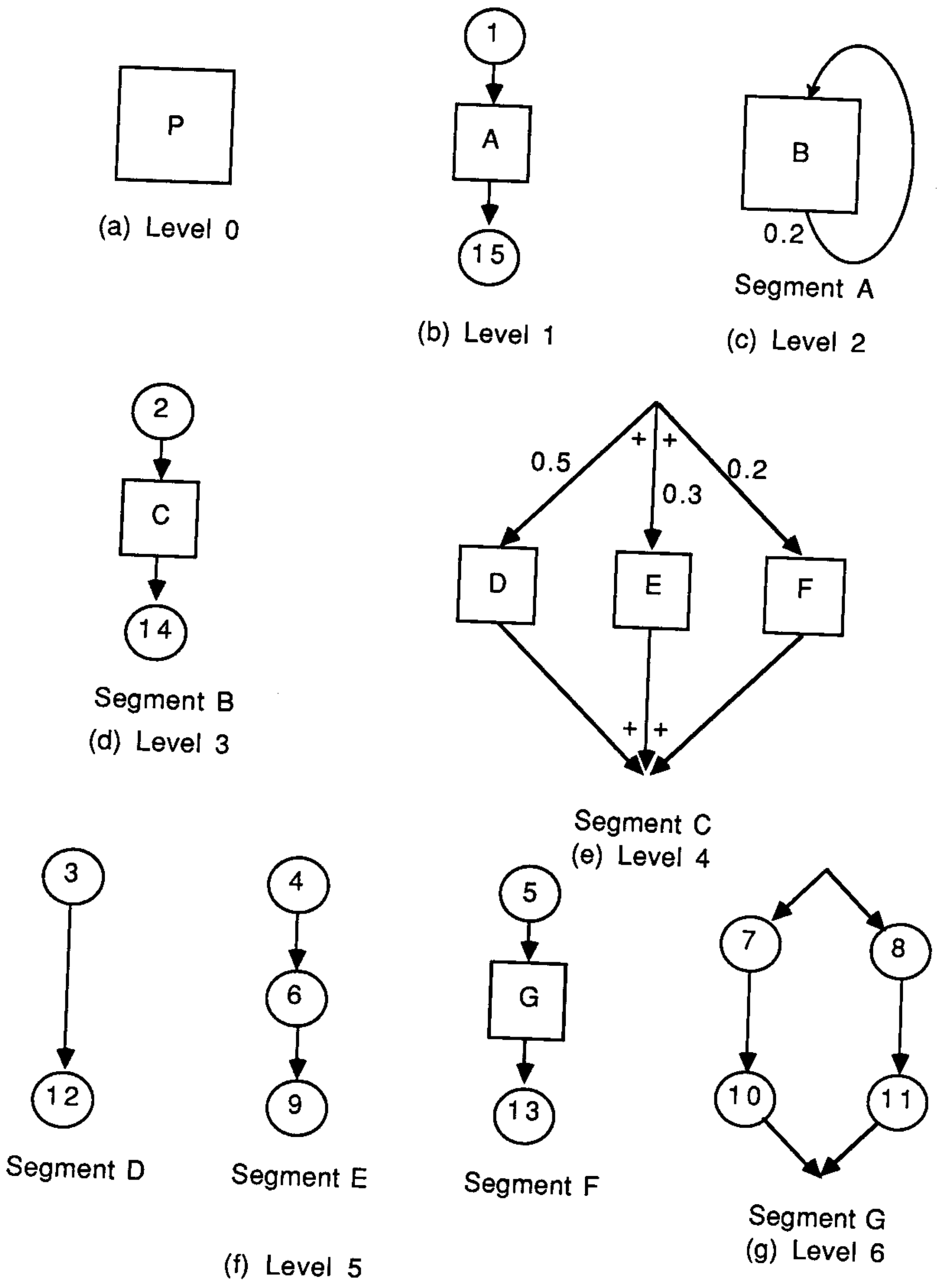


Figure 6.2 Hierarchical Decomposition of Process P

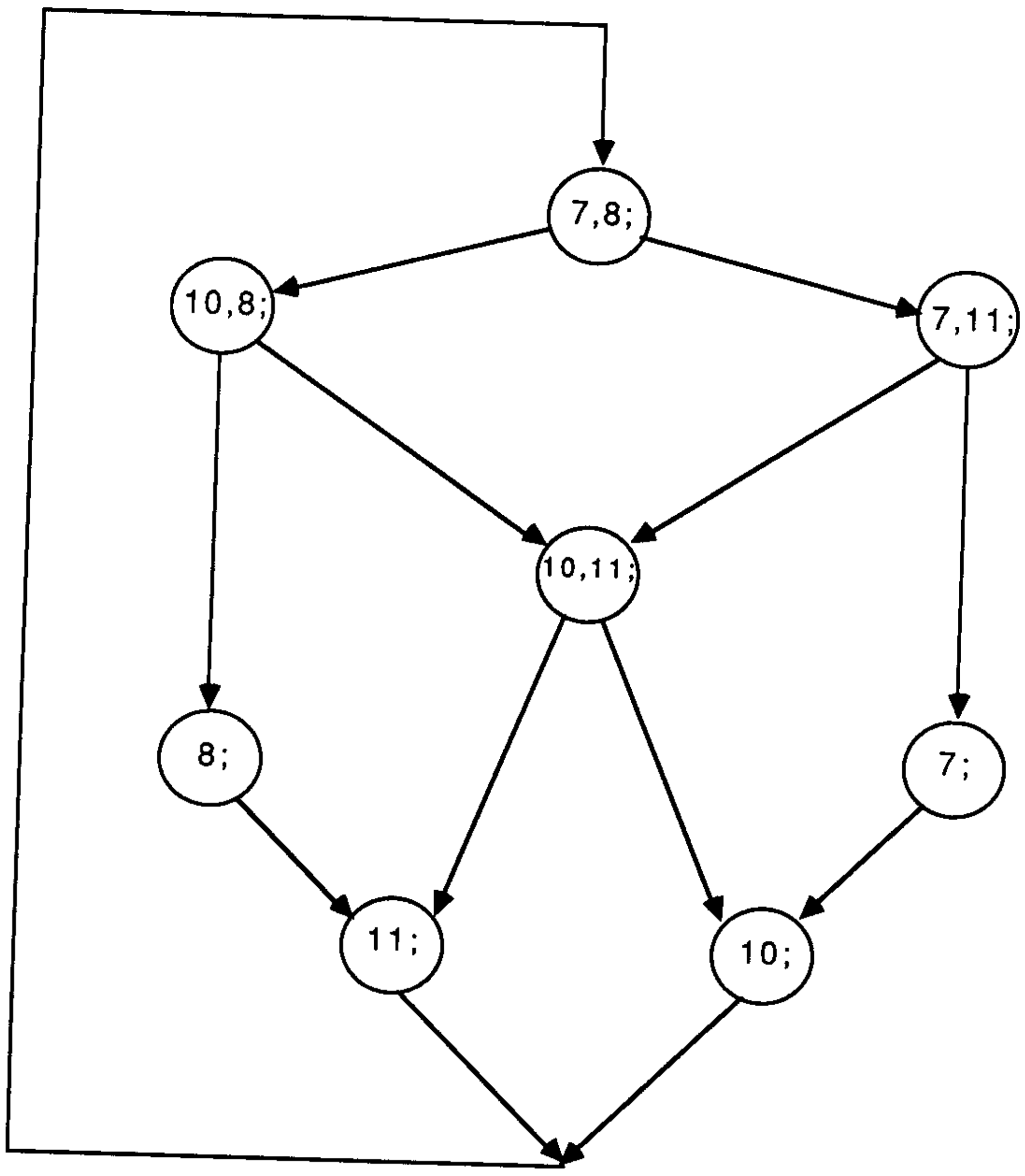


Figure 6.3 Markov Process for Segment G

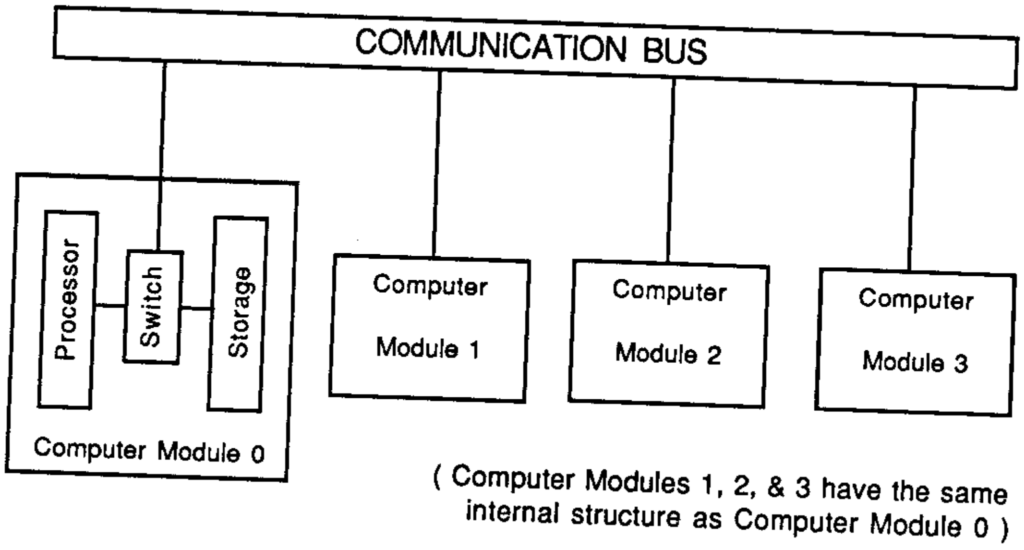
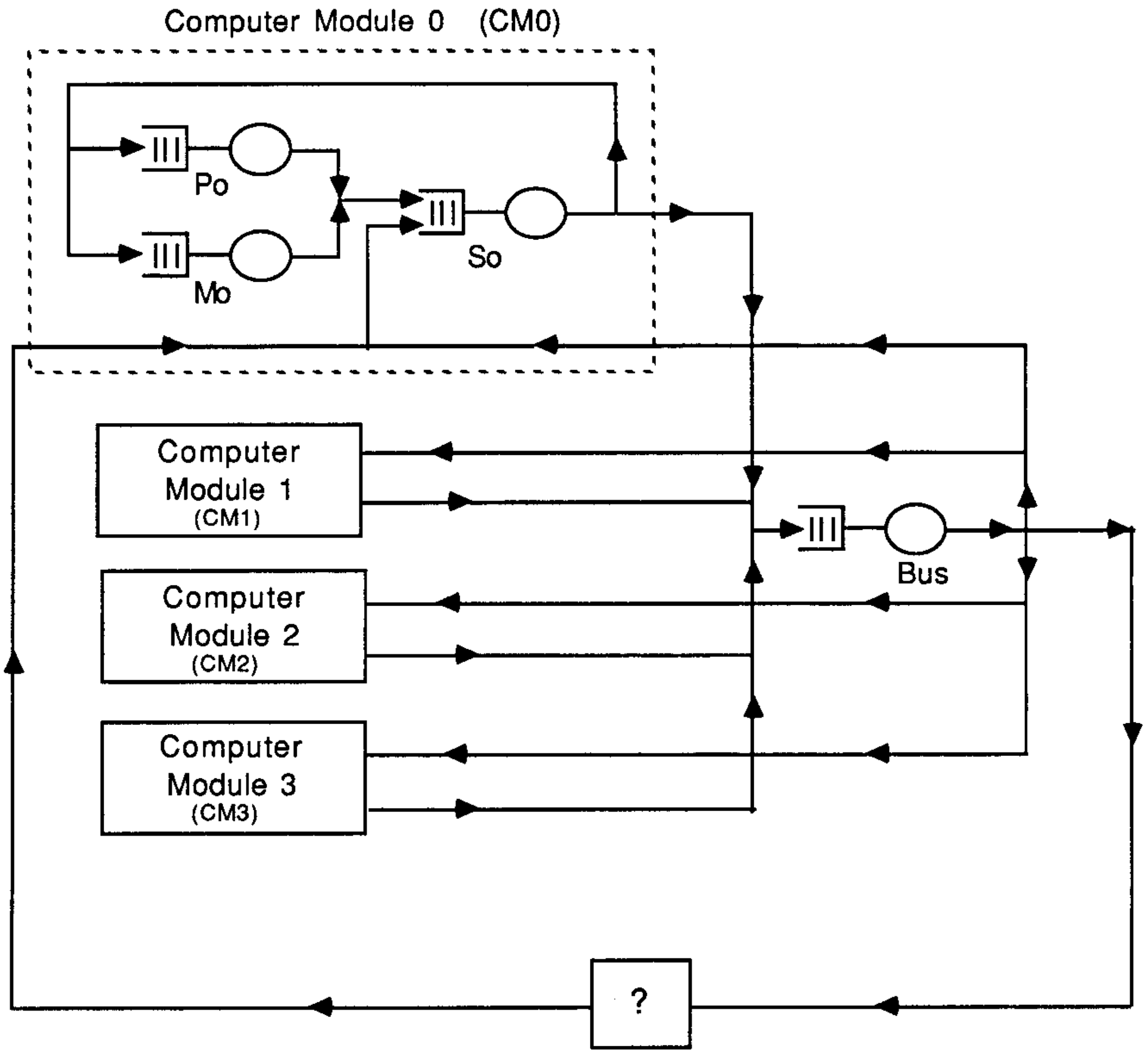


Figure 6.4 Architecture of the Execution Environment



(CM1, CM2, and CM3 have the same internal structure as CM0)

Figure 6.5 Physical Domain Model