THE UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

A TWO-LEVEL HIERARCHICAL CONTROL SYSTEM FOR A DISTRIBUTED NETWORK OF COMPUTERS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

degree of

DOCTOR OF PHILOSOPHY

By

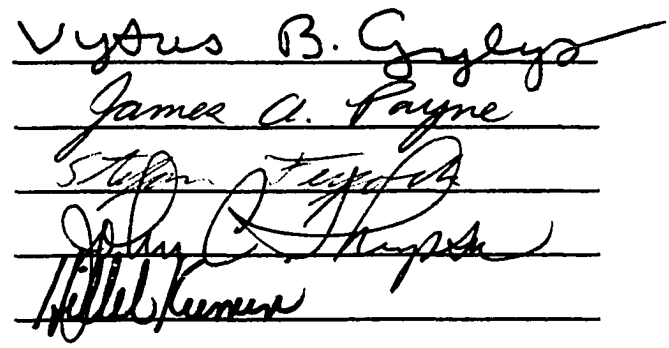
JUDITH A. EDWARDS

Norman, Oklahoma

1975

A TWO-LEVEL HIERARCHICAL CONTROL SYSTEM
FOR A DISTRIBUTED NETWORK OF COMPUTERS

APPROVED BY



DISSERTATION COMMITTEE

ABSTRACT

Control theory principles have been successfully used to determine operating policies for a variety of industrial processes. Hierarchical control methods offer the advantage of an adaptive system which is modeled by a top-down, modular approach. A two-level hierarchical system containing reconfiguration and local scheduling is demonstrated for a distributed network of computers connected by a common bus. The performance index on the optimizing level, reconfiguration, is to minimize the bus traffic by assigning the interacting programs to processors within the physical constraints of the processors and the constraints of an on-line, real-time system. The methods of quadratic binary programming and cluster analysis are compared for the suitability to compute real-time configurations. The local scheduling of periodic messages and programs to minimize the number of missed deadlines is considered for the direct control level. A simulation of the hierarchical process is presented in terms of the number of missed messages due to system overhead and failures.

ACKNOWLEDGEMENT

I would like to acknowledge the help and support of the committee who aided me during the preparation of my manuscript. I especially appreciate my co-chairman, Dr. Vytas B. Gylys, for his perseverance and for his direction. Appreciation is also extended to Professors James A. Payne and Hillel J. Kumin for their assistance in my research efforts. I also would like to thank Drs. Stefan Feyock and John C. Thompson for their comments on the manuscript. Sue Ann Lakey has been most helpful in the typing and the preparation of the manuscript.

CHAPTER THREE
A TWO-LEVEL HIERARCHICAL CONTROL FOR A DISTRIBUTED NETWORK OF COMPUTERS

CHAPTER FOUR
RECONFIGURATION TECHNIQUES

CHAPTER FIVE
SIMULATION OF THE TWO-LEVEL HIERARCHICAL CONTROLLER

BIBLIOGRAPHY

APPENDIX A QUAD and CLUSTER

APPENDIX B PROGRAM MODELS FROM APPENDIX A

APPENDIX C MODEL REPRESENTATION METHOD

TABLE OF CONTENTS

CHAPTER ONE
FUNCTIONAL REQUIREMENTS OF A
DISTRIBUTED NETWORK OF COMPUTERS

CHAPTER TWO
APPLICATIONS OF CONTROL THEORY CONCEPTS TO
ON-LINE, REAL-TIME COMPUTER SYSTEMS

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (continued)

LIST OF TABLES

CHAPTER 1

FUNCTIONAL REQUIREMENTS OF A DISTRIBUTED NETWORK OF COMPUTERS

In this study, control theory principles are used as an aid to software systems development. These methods were found to be suitable for designing operating systems for both multiprogramming and multiprocessing systems. To illustrate the applicability of the concepts developed in this thesis to actual computer systems, a multiprocessing case study was carried out and is described in the sequel. Multiprocessing environment was chosen for that study, because of the recent advances in technology that have made distributed networks of computers economically attractive [1.4.4]. Such networks have been widely used in command and control systems.

Distributed network configurations are ring, ARPAnet-like, common memory, or common bus arrangements, described in [1.4.1], [1.4.2], [1.4.3], and [1.4.6]. The network considered in this study is made up of independent computers, input sensors and IO devices connected by a common bus as in [1.4.1] and [1.4.6]. The main advantages of this system are: extendibility, lower hardware expenses, and greater reliability. The major design

difficulty has been found to be the traffic congestion over the common bus.

The hierarchical control concepts used in this study are highly intuitive notions that have been suggested in many areas that are only slightly related to the traditional control systems. The major attempts to present a formal theory of hierarchical control are found in works by Mesarovic [1.4.7] and Schoeffler [1.4.8]. These modular, subsystems of control are designed to meet the functional requirements of a distributed network, such as a command and control system.

For the model of a hierarchical control system for the distributed network, appropriate scheduling algorithms were found in the literature search; one of these algorithms was used to implement control on the local processor level. For the optimization methods to be used on the global level (i.e., higher than local) level of control, little research into algorithms was available in the literature which could be applied to the problem of physical constraints on the system.

Although many optimization algorithms are available in the literature of operations research, such as transportation and assignment algorithms, very few results can be applied to the physical problem considered in this study. These optimization methods rarely deal with a non-linear, binary programming problems with side-constraints. For example, the following algorithms typically do not have side-constraints:

polynomial assignment algorithms, generalized knapsack algorithm, and dynamic programming. Therefore, the algorithms, such as cluster analysis and quadratic binary programming, are investigated in this thesis.

1.1 Functional Requirements of a Command and Control System

Command and control systems were one of the earliest applications requiring parallel processing as well as distributed networks. These systems will serve as an illustrative example of distributed system requirements that can be met by using hierarchical control methods.

Porter, in [1.4.2], described the functional requirements of a command and control system. Examples of these systems are: air traffic control, ballistic missile defense, and industrial process control. Command and control systems are characterized by a large number of independent devices, inter-related tasks, and real-time response requirements. The functions of such a system are as follows: execute programs, service displays, respond to manual data insertion, and inter-processor communications. Porter also demonstrated that a multiprocessing environment could best meet these needs. However, at that time (1962), the state of the art dictated that independent processors share a common memory. With present technology, the costs of the network can be reduced by using independent, autonomous computing elements, each of which consists of a processing element, memory, and bus

interface unit, and by using a common bus that can easily be extended to handle more devices. Such a configuration also reduces the memory interference that is characteristic for a shared memory environment; it increases the vobustness of the entire system to local faults. A reduction in network communications reduces the operating system complexity.

The four functional requirements of a command and control systems according to [1.4.2] are availability, adaptability, expansibility, and programming criteria. Availability is a function of hardware reliability and maintainability. Porter in [1.4.2] states, "Costs of unavailability are higher in a command and control system...downtime for preventative maintenance is not permitted...some portion of the system must be available at all times." Graceful degradation of the system is permitted during system monitoring activities and during failures to computing elements, but the operating system must be able to bring the system back to full operation levels. Adaptability implies that the system can evaluate demands upon the resources and can adjust the priorities that are assigned to the individual tasks. The configuration must respond to important changes in the system. Expansibility must allow the command and control system to incorporate new functions, i.e., programs as well as processors. Programming criteria must be independent of the configuration, problem mixes, and task locations.

Lehman in [1.4.2] stated that the adequate performance

of such a parallel system is predicated on an appropriate low-level of operating system overhead. The fundamental functions of the executive control system is to perform allocations, scheduling, and monitoring, which must be simplified so that the total processing requirements of the executive is held to a minimum. Any gains from resource sharing and load smoothing must exceed the additional overhead due to resource allocations, conflict resolution, and other concurrent processing requirements.

These considerations suggest the use of multi-level, hierarchical control structures. For example, a 2-level hierarchical control structure considered in this thesis can reduce the operating system complexity and be designed to meet more than one performance requirement, such as the ones mentioned previously. The configuration control level is used to satisfy the functional requirements of adapting to the changes in the state of the system and its environment. This level will also allow changes in the processing functions and is processor independent. The scheduling level of control can minimize missed deadlines. The ability of the system to adapt to the state of the distributed system while reducing the amount of bus traffic between processes will be demonstrated by an example in the next section. It is important to note here that in some systems (especially in those without response time constraints) when the bus becomes congested the processing of some jobs may be delayed until the

bus traffic is relaxed. However, in command and control or process control systems, an increase in the bus traffic can lead to information losses. The following simple example is presented to illustrate the type of operating system resource allocation problem considered in the dissertation.

1.2 Reconfiguration Example

Consider a set of six interacting programs and their data bases as shown in the program schema of Figure 1.1, where p_1 is a program that produces a message of type m_1 and if the p_6 and p_3 cannot be assigned to the same processor, the amount of time required to transmit the message is 10 time units. Note that all six programs must be completed within a specified interval of time. Also, additional overhead to the operating system is incurred in preparing the message for transmission and decoding the interrupt. The program interactions are also given by the incidence matrix of Table 1.1, where the rows represent the programs and the columns represent the records (or messages), with the value 1 implying that the program uses that record (or message) and a value of zero otherwise. The message transmission time lengths on the bus are given in Table 1.1.

Table 1.2 shows the amount of run-time and the number of pages memory required to process each program in one of the six identical processors. It also states that the time interval available to process the entire set of programs is

Figure 1.1 Process Schema indicating program-message interaction and precedence of the message traffic.

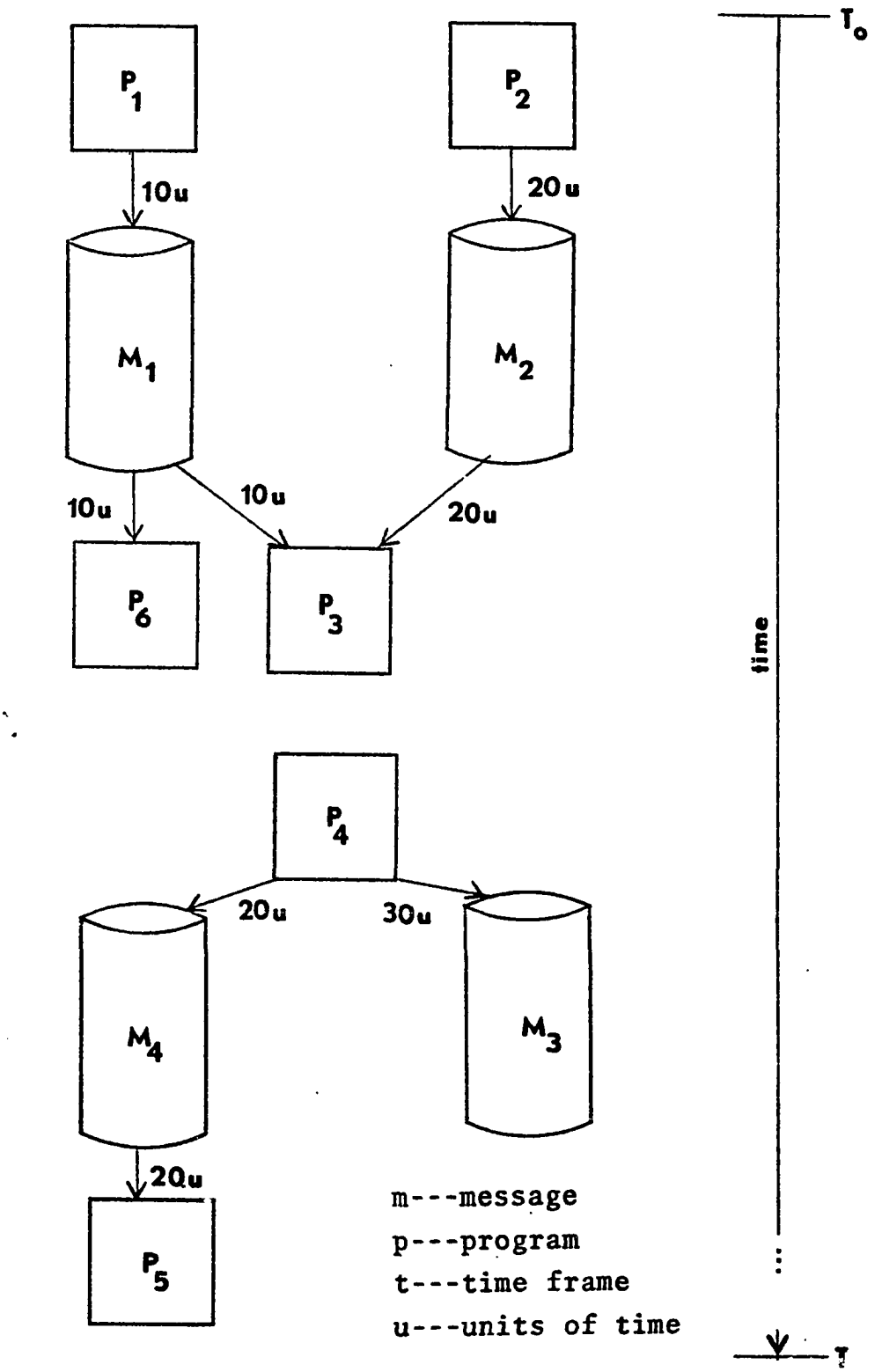


TABLE 1.1 Message - Program Interaction

(State Table)

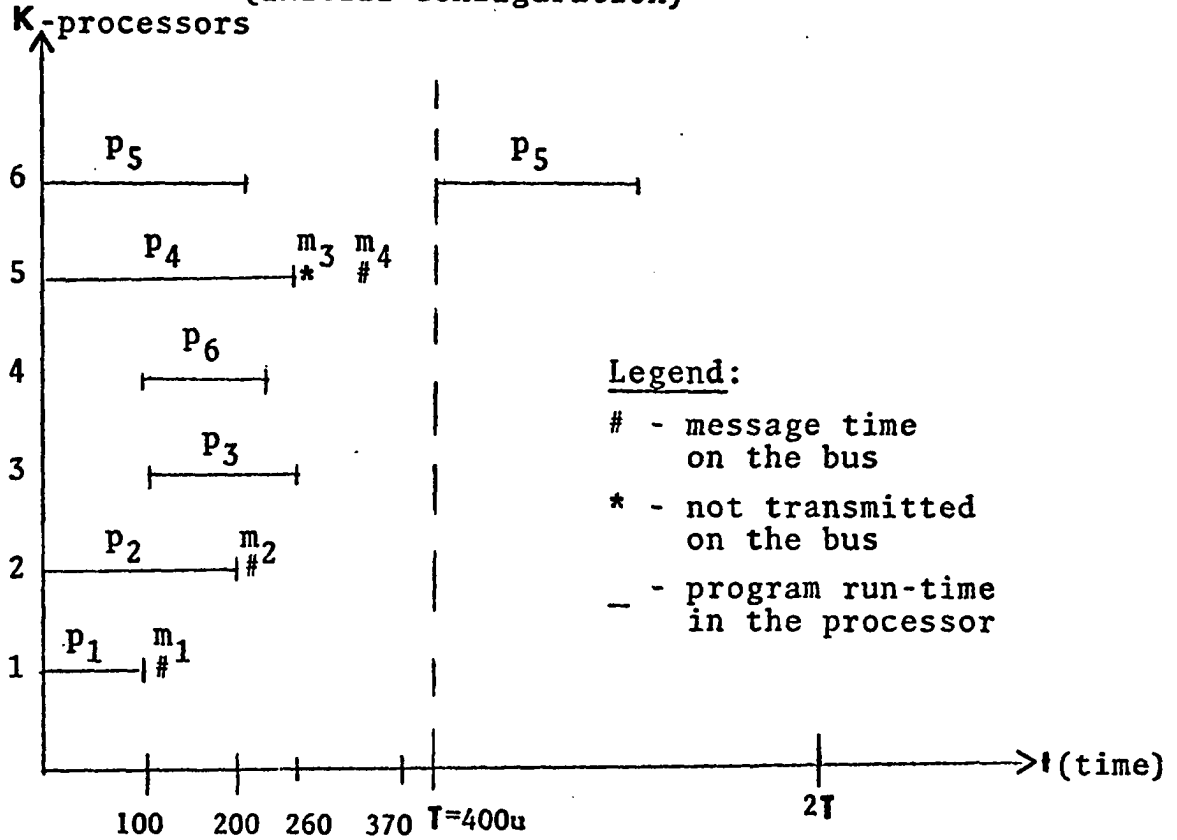
Programs \ Messages	1	2	3	4
1	1	0	0	0
2	0	1	0	0
3	1	1	0	0
4	0	0	1	1
5	0	0	0	1
6	1	0	0	0
Length of message transmission time (units)	10	20	30	20

TABLE 1.2 Program Description

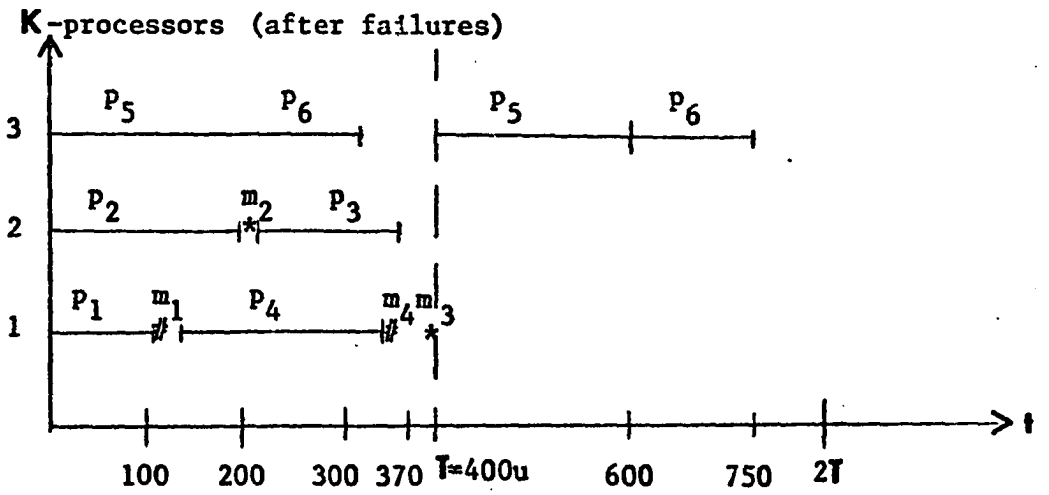
Program Identification Number	Run-time Per Frame (Units)	Pages
1	100	1
2	200	2
3	150	1
4	250	2
5	200	1
6	150	2

Available processing time = 400 units
Memory per processing element = 3 pages

Figure 1.2 Gantt Chart
(a) 6 processors and 1 program per processor
(initial configuration)



(b) 3 processors and 2 programs per prccessor
(optimal solution)



400 time units (which is the length of cyclic time) and that 3 pages of storage are available per processing element.

The Gantt chart in Figure 1.2a shows the initial schedule of the six programs with respect to the six processors. Program p_1 produces message m_1 and programs p_6 and p_3 are located in separate processors which cannot schedule these programs until message m_1 has been transmitted. The Gantt chart depicts the earliest possible times the programs can be scheduled to complete the execution of the programs within a standard time frame.

Suppose that three processors fail. The reconfiguration problem becomes the problem of assigning the six programs to the three processors in order to minimize the amount of bus traffic over the common bus. Failure to minimize the bus traffic may cause the programs to miss message deadlines and thus to lose some messages altogether. Table 1.3 has the possible assignment of all pairs of programs to the same processor, e.g., programs p_1 and p_2 can be assigned to the same processor and utilize 300 units of the major time frame and utilize 3 pages of memory, but the amount of interaction time between the two programs is zero (there are no messages shared). In the case that the programs cannot "fit" within the processor, a dotted line is used as in the assignment of programs p_2 and p_4 which would use 450 units of time and 4 pages of memory, both being greater than the processor capacity.

TABLE 1.3 Reconfiguration Program

Assignments to Processors

Assignment Program Identification	Total Runtime	Total Pages	Message Interaction Time
1,2	300	3	0
1,3	250	2	10
1,4	350	3	0
1,5	300	2	0
1,6	250	3	10
2,3	350	3	20*
2,4	——	——	——
2,5	400	3	0
2,6	——	——	——
3,4	400	3	0
3,5	350	2	0
3,6	300	3	10
4,5	——	——	——
4,6	——	——	——
5,6	350	3	0

* Maximum Interaction
—— Unable to assign due to processor constraints

TABLE 1.4 Processor Assignment

		Programs		
	Processor 1	2 & 3*	1 & 6	1 & 4
	Processor 2	1 & 4	2 & 5	2 & 5
	Processor 3	5 & 6	3 & 4	3 & 6
Total inter processor communication time	Time Required	50	60	60
	Time Saved	20	10	10

* Optimal solution

Table 1.4 contains all feasible allocations of the programs to the processors, but the first column in the table is the optimal solution. In this tightly constrained case, the maximum bus traffic savings in the transmission time are 20 time units. The Gantt chart of Figure 1.2b demonstrates a schedule to execute the programs in order to meet the message traffic deadlines and not cause further loss of information.

The bus scheduling and the synchronization of the message transmission and program execution can be relaxed if double buffering is adopted. In this case, two messages can be available to each processor before one of the messages is overwritten by the transmission of a third message. Methods used to compute the reconfiguration and the schedule are considered in later chapters. The 2-level hierarchical control scheme allows the system to continue processing in a periodic fashion until a change in the availability of one of the processors occurs. The overhead due to performing the reconfiguration and scheduling can cause further loss of messages if the resulting schedule cannot be processed within the allotted time frame T. A simulation of the example presented in this section and other more comprehensive models are presented in the last chapter.

1.3 An Overview of the Chapters

The background information related to the control theory concepts and hierarchical control systems is summarized in

Chapter 2. Several types of models used in control theory systems are also reviewed there. A wide variety of problems approached by means of control theoretic methods have turned out to be successful, which was one of the motivating factors in considering their application to operating systems. One known example of such an operating system, which is defined as an adaptive controller, is discussed.

Chapter 3 discusses the model of an operating system considered for a distributed network of computers; this model is structured as a hierarchical control system. The mathematical model formulation of the reconfiguration problem is presented as the top level of the hierarchical controller. One of the main functions of this controller is to adapt the network to changes in the network or in its environment. Two algorithms which were found in the literature search and which meet the scheduling requirements for minimizing the number of missed deadlines are presented as a possible subsystem in the hierarchical controller.

The reconfiguration algorithms for assigning programs to processors are outlined in Chapter 4. The implicit partial enumeration method is used to determine the optimal solution that will minimize the amount of bus traffic. Since this method may not meet the real-time processing requirements of the network, another method, which is based on cluster analysis, is used to compute a sub-optimal solution. A comparison of the methods is also presented. No literature

references could be found which would indicate whether these algorithms could handle the physical constraints on the assignment locations (the processors) in real-time. (On first reading, this chapter may be omitted without a loss of continuity.)

Chapter 5 discusses the simulation of the hierarchical controller for a set of models of distributed networks. The discussion contains the effects of the real-time reconfiguration and scheduling overhead upon the bus traffic. This overhead along with the amount of message traffic and the number of missed messages is given with respect to the number of processors available during the major time frame.

Procedures to perform the reconfiguration are contained in Appendix A. Tables representing the models of the process schema, similar to the example presented in this chapter, are located in Appendix B. All application programs are written in PL/1 to make use of dynamic storage allocation features and the wide variety of data structures available to the user. A "Petri net" discussion and diagram of the events in the simulation are in Appendix C. For ease of reading, local references are presented at the end of each chapter, e.g., [1.4.5] is the fifth reference found in Section 1.4 of Chapter 1. A comprehensive list of references is contained in the Bibliography.

This study assumes the availability of a master bus controller, which is not discussed. Other bus configuration

models could have been considered. The primary goal in this dissertation is to illustrate how the hierarchical control design can meet the functional requirements of a distributed network of computers.

1.4 References

1. Anderson, G. A., "Interconnecting a distributed processor for avionics." First annual symposium on Computer Architecture, 1973.
2. Bell, C. G. and Newell, A., Computer Structures; Readings and Examples. McGraw-Hill, 1971.
3. Farber, D. J., Feldman, J., Heinrich, F. R., Hopwood, M. D., Larson, K. C., Loomis, D. C., and Rowe, L. A., "The distributed computing system." Proc. COMPCON, February 1973, pp. 31-34.
4. Goodell, W., "The Minis Gang Up." Computer Decisions, vol. 6, no. 6, June 1974.
5. Hansen, P., "Programming methodology for operating systems design." IFIP (proc.) 1974, p. 394.
6. Jansen, E. E., "A distributed function computer for real-time control." Second annual symposium on Computer Architecture, 1973.
7. Mesarovic, M. D., "Multilevel systems and concepts in process control." IEEE (proc.), vol. 58, no. 1, January 1970, p. 111.
8. Schoeffler, J. D., "On-line multilevel systems." Wismer, D. A. ed., Optimization methods for large-scale systems. McGraw-Hill, 1971.

CHAPTER 2

APPLICATIONS OF CONTROL THEORY CONCEPTS TO ON-LINE, REALTIME COMPUTER SYSTEMS

Difficulties in designing and debugging operating systems have been apparent since the advent of third generation, multiprogramming systems. The system complexity, extendability, reliability, and overhead are the major problems encountered with the operating systems. Multiprogramming system studies have been well covered in the literature, but the problems associated with the design of distributed computer networks are new. In this study, control theory concepts will be presented as a method of designing an operating system for a distributed network of computers.

Well known control theory principles have been applied to biomedical, aerospace, and industrial processes with a great amount of success. This success is primarily due to the well-developed mathematical models (e.g., differential equations) for the control system dynamics. Control theory with its specialized areas of state space identification, performance evaluation, reliability, and stability can also be used in the design and modeling of software systems; in

particular, operating systems. To reduce the complexity and the overhead of the system, hierarchical structure for control systems are introduced to satisfy these systems goals.

The observability and controllability of a computational process controlled by an operating system are the main objectives of the control theoretic approach. This chapter defines a hierarchical control system and the controllers used in it. These concepts are applied in the next chapter to define a two-level hierarchical control system for a distributed network of computers. The performance of such a system is described in the chapter on simulation.

The state space approach has come from automata, graph theory (especially, Petri nets and macro E-nets), and stochastic processes. The NASA study by Northouse and Fu [2.5.7] is known as the most direct application of these ideas, the adaptive-feedback control principle, to a multi-processing operating system. The model introduced by Northouse and Fu is reviewed in this chapter because of its direct bearing on this research.

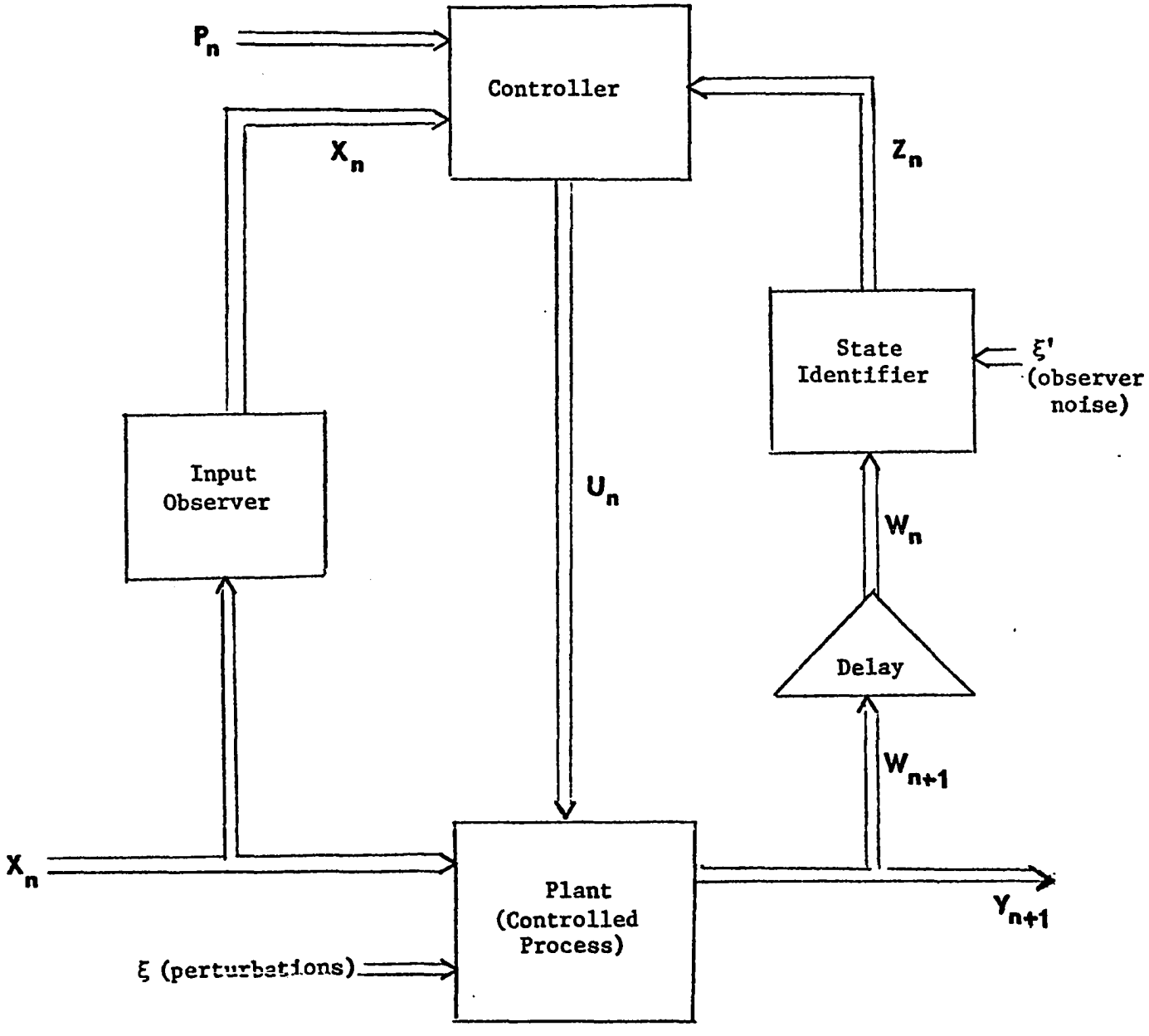
2.1 Relevant Control Theory Concepts

The next objective is to set up a control theoretic framework for the analysis and design of operating systems. To state a control problem rigorously it is necessary to define the relevant concepts, such as system environment,

controller, process, or system state. General references for these definitions are: [2.5.1], [2.5.2], and [2.5.4]. Figure 2.1 shows the system interactions in a generalized control system. The "information" vectors are associated with the modules of the control theoretic examples in this chapter and in the next chapter and are defined in this section.

Consider a "process", sometimes referred to as the plant, that is to be controlled. This process can represent the behavior of a physical device, such as a car, or a system procedure. The control mechanism, or controller, is essential to accomplishing a goal for the process, e.g., to prevent chaos, to improve efficiency, to increase production, or to coordinate production. For a computer system, the controller (operating system) is a computing process which utilizes the hardware, software, and data (stored information) resources and controls other computing process sharing the same resources. The system environment of the process is represented by the external process inputs and the computed outputs fed back into the environment. In a general purpose computing environment, the inputs as well as the outputs are the data and the job stream. (The process inputs are not necessarily the same as the outputs, i.e., in manufacturing processes the inputs are the raw materials and the output is the finished products.) These outputs often arise as a response to the system environment. Therefore,

Figure 2.1 Generalized Controller



the system under consideration consists of the controller, the controlled process and the interface between the environment.

To define the "information" vectors, shown in Figure 2.1, the following equivalent notation is used to represent a vector X at time $t=t_n$: $X_n=X(t_n)=X(n)$. To distinguish between function and product for f(t), an asterisk (*) is used to denote multiplication, i.e., f*t. Vectors are capitalized and the vector components are lower case with subscripts, e.g., $a_i(k)$ is the i^{th} component of A at time $t=t_k$. Let the zero subscript following a vector symbol, denotes the initial value of that vector, i.e., X_o represents the initial value of the vector X.

Let an ordered 6-tuple represent the "system space" which contains the "information" vectors defined over the time interval, $T=(t_o, t_n)$, under consideration; thus, R(t)=(X(t), Y(t), Z(t), U(t), W(t), T). The external process inputs (or simply the input variables) at time t are represented by X(t) and Y(t), respectively. The controller may not use all of the processed outputs, but may only use a "sample" of the outputs, shown in Figure 2.1 as W(t), which often are called the observables. The sampling period may involve a finite amount of delay, also shown in Figure 2.1, before the information is reported to the state identifier. The output from the controller is U(t) and is referred to as the control policy (also the control vector or control

variables). The static description of the control environment is often referred to as the control parameters, P in Figure 2.1. These control parameters are exogenous inputs to the controller. For a computer system, the control parameters contain information concerning the hardware configuration, e.g., the number of processors, IO device descriptions, memory size. These control parameters are contained in U.

One of the most important concepts in the system is that of the state vector, or of the state of the system. An observable portion of vector Y(t) (i.e., a function of X(t)) is the state vector, Z(t), containing information on the status of the system such that knowledge of Y at any time instant t in T is sufficient to fully describe the system at that time t. In the applications under consideration, it is convenient to restrict the class of system models to that of discrete systems. For computer systems, the status vector Z(t) of the system would contain information on job queues, supervisor tables, status of IO devices, and current control policies.

Since the system environment under consideration is not completely known, stochastic noise (or perturbations) is often introduced into the system model to account for the unknown portions of system dynamics. This interference is shown in Figure 2.1 as ξ for the processing noise and ξ' for the sampling noise.

The system dynamics are modeled by a state transition function, i.e., Y(n+1)=funct(X(n), ξ(n), U(n), T). Control theoretic applications have been successful primarily due to the extensive results available for modeling the system dynamics, e.g., systems of deterministic or stochastic differential equations, from which the control policies can be computed. Discrete models of systems, such as computational processes are not as well-developed; however, the use of stochastic differential equations (i.e., the diffusion equation applied to queueing models) has had limited success in modeling such discrete systems.

A system is said to be controllable if there is a control policy, U, which produces acceptable values Y in finite amount of time. Observability implies that by observing the outputs, Y, during a finite interval of time, say (t_i, t_j), the input, X, can be determined [2.4.1].

In some models, the system is required to meet certain performance criteria. A performance index is a real, scalar objective function of the process variables, i.e., J=funct(U(t), Y(t), t). Typical performance indices for computer systems have been throughput, turnaround time, response time grade of service, etc. The type of model determines the role that the components of the control system will have on the processes within the system.

2.1.1 Types of control models

As noted in [2.5.2], there are essentially four types of control models, not necessarily mutually exclusive, which are used to view and design controllers. These basic models are known as the process, procedural, predictive, and optimizing models. Such a model provides guidelines which assist in evolving a system design. The application of a particular model may result in a control system that would differ from one based on another model.

A process model considers the mathematical and logical relations among the control variables, the state variables, and the performance criteria. As noted in the preceding section, the performance index (criterion) normally is a function of the control and state variables.

A procedural model is a process model which explicitly specifies the timing relationships among the process variables and the conditions (threshold values) triggering various actions. The timimg relationships between two successive states of a system may be expressed by means of a state transition function; for example,

$$Z_{n+1}=\text{funct}(Z_n, U_n, t_n)$$

where Z_n is the system state at time $t=t_n$ and U_n is the control applied at t_n. An example of a procedural model for a channel processors is the completion of an IO task which triggers an IO interrupt (changing a state value) that

requires a different procedure to handle the condition (the interrupt).

A predictive model is a procedural model which uses the information about the present and past states of a system and the previous estimates to forecast the future outcome of the process. Such an estimate of the process future may in time be utilized by the controller to improve the control policy.

An optimizing model is a process model which utilizes an objective function (performance index) to compute good policies. An optimizing model does not necessarily show time relationships. For example, in some resource allocation problems the state vector is not an explicit function of time. An optimizing model may or may not have predictive capabilities. The optimizing model seeks an operating point (solution to the control problem) that maximizes or minimizes the index of performance within the constraints placed upon the model. Often the optimizing model cannot be used due to the mathematical formulation of the control problem (there is either no solution or the solution method takes too long to be suitable for real-time control) and suboptimal operating points are produced by heuristic methods.

The control problem considered in this study is to assign and schedule programs to the processors in the distributed network. The performance criteria used is to

minimize the bus traffic among the distributed processors and minimize the number of missed messages.

2.1.2 Rules for choosing performance index

In classical control theory, the index of performance should be chosen so that the following criteria are satisfied:

1. there is a set of controls that will optimize the objective function: if more than one set of points exist in the domain, then there are alternative policies.
2. there must be an appropriate choice of cost coefficients for combining measures of different physical entities. (Errors occur in interpreting the performance index that do not represent a physically meaningful quantity or in providing resolution among the control policies.)
3. there will be only one objective function for each model.

Often the system has conflicting performance criteria. The composition of such a system may lead to further difficulties in determining the control policy. In order to reduce the complexity of the model, a system of interacting sub-models will be defined as a hierarchical control structure and will be presented in a later section. These sub-models may have different goals and different methods of control.

2.1.3 Control model refinements

Controllers are said to be time-varying if the state equations can be written as: $Z(t_i)=\text{funct}(Z(t_{i-1}), U(t_{i-1}),t)$.

If the state equations are written: Z(t)=funct(Z(t), U(t)), then the system is said to be time-invariant. Another classification of control is into open-looped and into closed-looped forms (see Figure 2.2). Open-looped form implies that there is a function of the state variables at initialization time, t_o, and time, t, which represents the optimal control policy, i.e., $U^*(t)=funct(Z(t_o), t)$. For the closed-looped form, (also referred to as optimal feedback control, optimal policy, and optimal control strategy), the optimal policy would be a function of the state variables at any time t, i.e., $U^*(t)=funct(Z(t))$.

In addition to the control models presented previously, there are control systems in which all of the process variables are not known. Models of these systems have been used to allow the control policy to "adapt" to the state of the system and to "learn" to estimate the unknown process variables. For example, consider the linear programming problem:

$$\text{optimize} \quad z = C \cdot X \qquad \text{(eq. 2.1)}$$
$$\text{subject to} \quad A \cdot X \leq B$$

where A, B, C, and X are matrices of dimensions mxn, mx1, 1xn, and nx1, respectively. Assume that the linear programming problem expressed by eq. 2.1 is a control system for some process in which all of the parameters are not known, say A. One such model of this system would allow the control policy, U, e.g., a solution to eq. 2.1 to "adapt" to

Figure 2.2a Open-loop optimal control

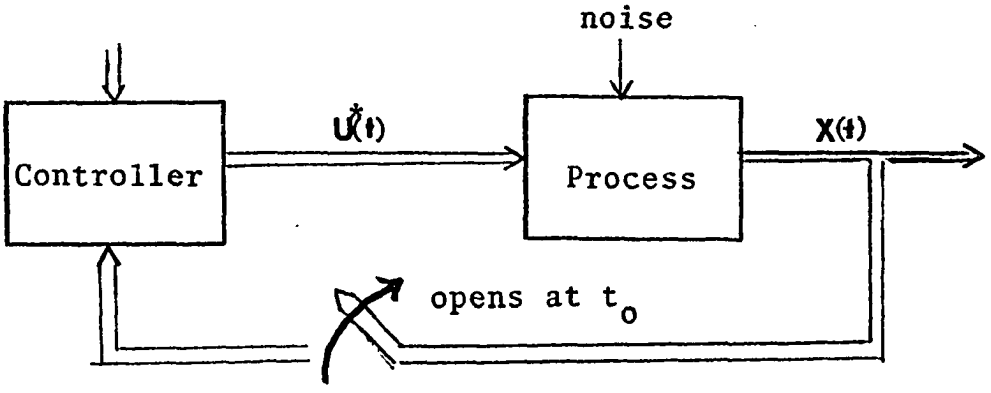
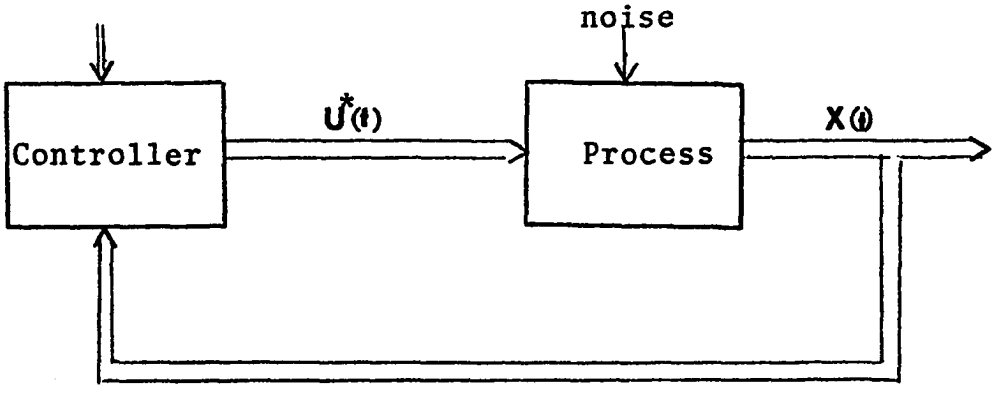


Figure 2.2b Closed-loop optimal control



the state of the system and to "learn" to estimate the unknown process variables, such as the values in A.

Concepts of these models are highly intuitive. There has been attempts to arrive at rigorous definitions for self-organizing, adaptive, and learning systems. The following definitions of these terms were proposed by G. N. Saridis, J. M. Mendel, and Z. J. Nikolio in [2.5.3].

A control process is called self-organizing if reduction of the a priori uncertainties pertaining to the effective control of the process is accomplished through information accrued from subsequent observations of accessible inputs and outputs as the control process evolves. This allows the control system to change by adding (or deleting) control variables as the effect of the control policy is evaluated. For example, let $A=(a_{ij})$ in eq. 2.1 where a_{ij}= original input, if the variable x_i is to be used in the control problem and $a_{ij}=0$, otherwise; for some i and all j. To reduce the dimension of the linear programming problem the values in C would have to be defined in a similar manner.

Two basic types of adaptive control are known. A parameter adaptive, self-organizing control process implies that it is possible to reduce the a priori uncertainties of a parameter vector characterizing completely the process through subsequent observations of the accessible inputs and outputs of the system as the process evolves. In the example of the linear programming problem, the values of A, B, and C

are the parameters that may be altered as the performance of the system is evaluated, e.g., $A=f_k(A, \text{predicted } z, \text{actual } a)$ where k is the number of times the control policy has been produced. A performance adaptive, self-organizing control process implies that it is possible to reduce the a priori uncertainties pertaining to the improvement of the performance through subsequent observations. In the case of the linear programming problem, there can be more than one objective function (i.e., performance index) as the process evolves. For example, let C_1 and C_2 be cost coefficients, then

$$C=\begin{cases} C_1 & \text{if } \tilde{z} \leq Z_o \\ C_2 & \text{if } \tilde{z} > Z_o \end{cases}$$

where Z_o is a threshold value and $\tilde{z}$ is the actual performance resulting from the control policy.

A learning control process uses acquired information to control a process with unknown features. In eq. 2.1, the values of A can be a weighted average of the previous values, depending upon the success (or failure) of the control policy. Let $A_k=f(A_{k-1},p,q,z,\tilde{z},t)$ where p,q are scalars that are averaged into the old value, then such an average function, f_k, is given by

$$a_{ij}(k)=\begin{cases} \dfrac{(k-1)*a_{ij}(k-1)+p}{k} & \text{if } |z-\tilde{z}| \leq Z_o \\ \dfrac{(k-1)*a_{ij}(k-1)+q}{k} & \text{if } |z-\tilde{z}| > Z_o \end{cases}$$

for a threshold value Z_o, and p is an enhancement factor and q is an inhibit factor. Learning control may be performed off-line in which case the controller is trainable or on-line in which case it is synonymous with the performance adaptive, self-organizing control.

These concepts as presented in [2.5.3] will be used to describe the closed-loop, adaptive controller for a multiprogramming system that was designed by Northouse and Fu and the adaptive component of the two-level hierarchical system.

2.2 Hierarchical Control Structure

There are two reasons for choosing hierarchical control structure for the operating system for a distributed network of computers. First, the control problem depicted in the network in Figure 2.3a is conceptually easier to solve than the one in Figure 2.3b. This could well account for the advantages that Dijkstra noted in the top-down design used in "THE" operating system which resulted in reducing the number of mistakes in coding and in completing on time with fewer programmers [2.5.17]. This was the primary motivation for structuring the operating system for the distributed network of computers.

In this section, the characteristics of a hierarchical, or multilevel, control system will be presented. The modules that make up each level are controllers. There are examples of several systems in which it is possible to link

Figure 2.3a Decomposition into Hierarchical Control System

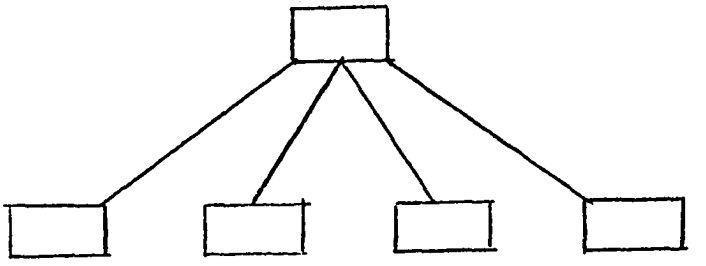
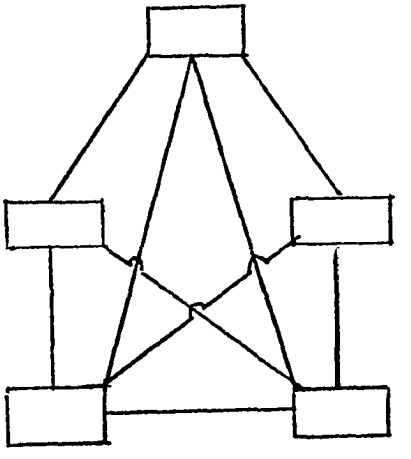


Figure 2.3b Network of Computers with one Processing Element Acting as Control Computer

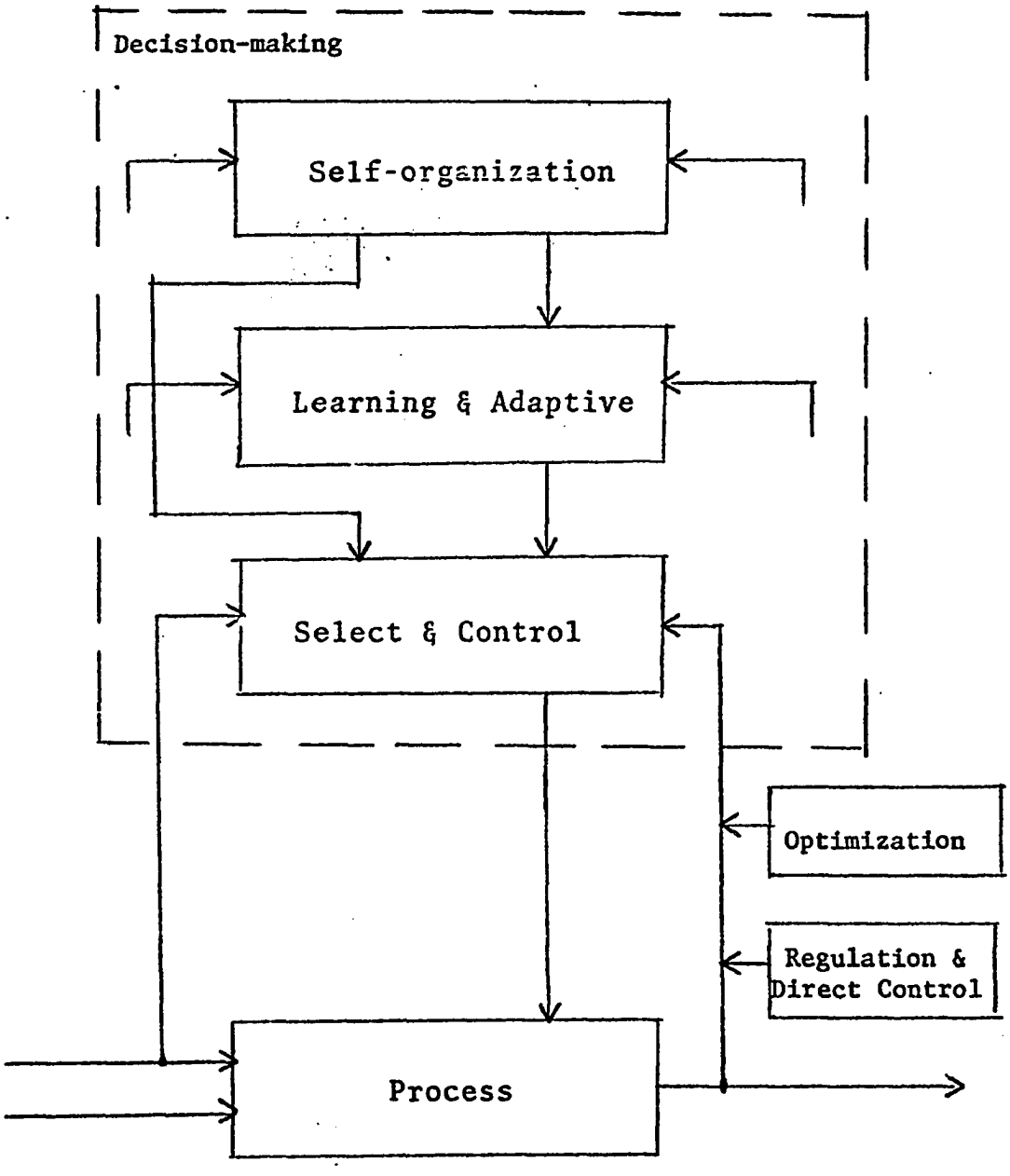


various types of control subsystems [2.5.6]. For example, the less frequently executed module such as a self-organizing level, may be the top level with the optimization level being the next sublevel and the direct control as the bottom level (Figure 2.4). Advantages and disadvantages of using hierarchical control are also presented at the end of this section.

In Chapter 3, a two-level hierarchical control system is defined for a distributed network. One level is a self-organizing parameter adaptive control system used to reconfigure the system when there is a change in the number of active processors. The second level is an optimizing and selection control system to schedule programs within the processors. The performance indices for the control problems are the amount of intra-processor communications and the number of missed deadlines.

Primary applications of hierarchical control systems have been for industrial process control [3.5.15 & 3.5.16]. Schoeffler, in [2.5.6], states three different methods of modeling a hierarchical control system, i.e., decomposing the control problem into sublevels based upon the levels of physical structure, the levels of influence, or the levels of control. The decomposition of the control problems on the basis of physical structure implies a partitioning of the model in terms of physical entities, e.g., time or space. The levels of influence are analogous to a decomposition in

Figure 2.4 Hierarchical Control



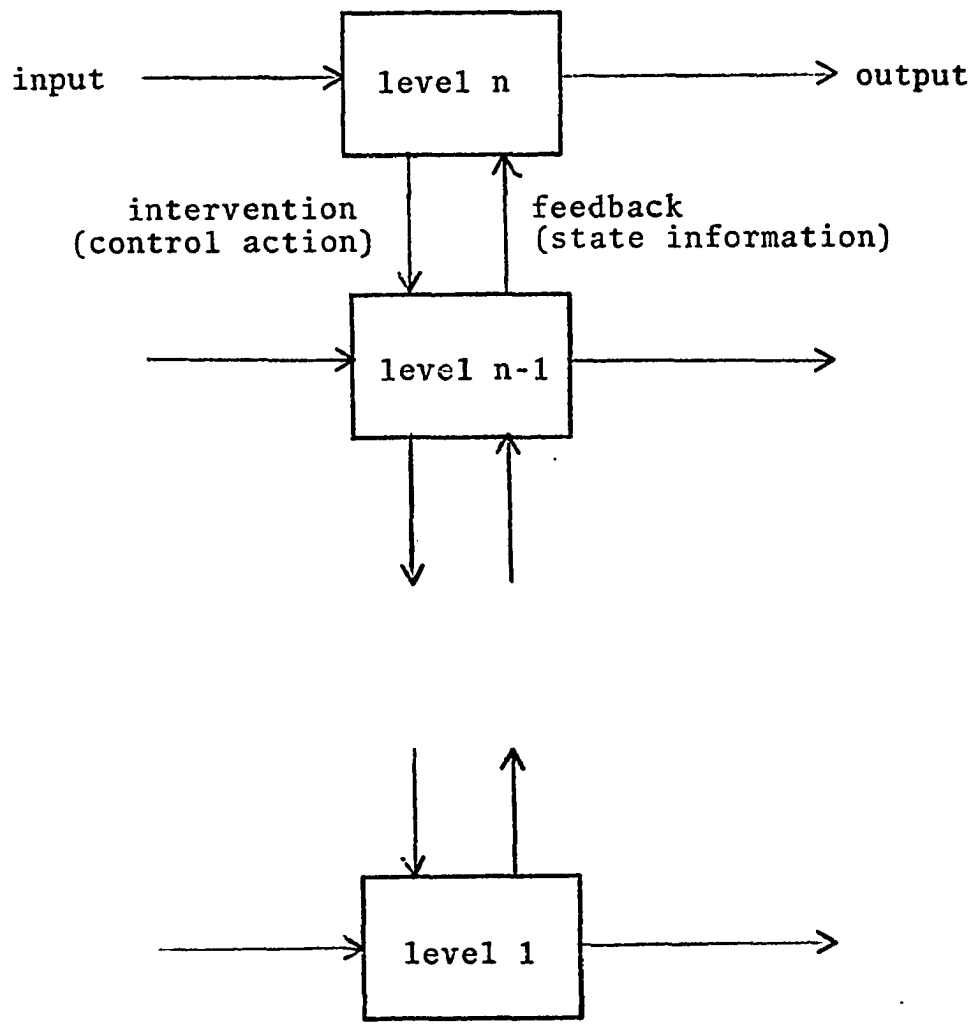
a business organization chart or to the levels of access in a management information system. Examples of each method are given in [2.5.6], but the last method (levels of control) is the most common and is the one used in the two-level operating system model. Decomposition on the basis of level of control places direct control at the lower levels and more expensive, less frequently executed, levels are at the higher levels.

2.2.1 Characteristics

A multi-level hierarchical structure is not a well-defined model by current standards in control theory; therefore, Mesarovic, attempting to establish a formal theory of hierarchical control, only presented the characteristics of such a control system in [2.5.5]. These three characteristics are vertical decomposition, "priority of action", or "right of intervention", and (vertical) performance dependence. In Figure 2.5, the vertical decomposition is a sequential order of subsystems with the input of one level being the output of the preceding (or higher) level(s), e.g., a higher level passes control information to a lower level. The information from a level can be distributed to all levels.

The transformation of the "information" vector input into a sublevel can either be a dynamic or a periodic sampling process as well as a problem solving procedure.

Figure 2.5 Generalized Hierarchical Control



The priority of action, control that is oriented downward as in Figure 2.5, results from the higher level influencing the operation of a lower level. Usually this action only influences the immediate successor subsystem. The performance dependence of the model implies that the intervention (in the form of commands) is passed down to lower levels and that information is fed back (in the form of status information) to the higher level.

Using the level of control method of decomposition for the operating system, those systems that cause the greatest change in the total system definition or that require expensive procedures (in terms of execution time, size, and/or complexity) are the upper levels of control. The highest level of control affects the process parameters such as control coefficients, A, B, and C in the linear programming problem 2.1, and is an adaptive layer. The second (or middle levels) is the key control which affects processes or optimization, i.e., changes the status of the system Z or produces the optimal control policy U. And the lowest level represents direct control where most process disturbances affect the process variables, or the values of the control variables, Z.

2.2.2 Inherent problems in a real-time, on-line system

There are many problems in designing a control system for a real-time, on-line problem. Difficulties of the

following three types arise due to the problem complexity. First, the class of disturbances may be very broad. This suggests that alternatives may have to exist to solve the problems that arise. Secondly, an appropriate structure or configuration may not be self-evident except in simple cases. Even if the structure is known, it may be overwhelming to implement the total system. Thirdly, real systems are dynamic. Over a period of time, disturbances may arise that were not apparent at the beginning of the design and may require a change in controls.

Difficulties of still another type have to do with the implementation of the design. Each portion of the control system may require different data. Poor structuring may result in large amount of interprocessor communication, decreasing realiability, and increasing costs. The control strategy can pose major programming difficulties and debugging and system interface problems. Hierarchical control was suggested by [2.5.5] and [2.5.6] to create a dynamic, modular system that will avoid some of the design and implementation problems in real-time, on-line systems.

2.2.3 Advantages/Disadvantages of a hierarchical design

The advantages of a modular system arranged in hierarchical control structure are:

1. add to (delete from) the model without redesigning the system.

2. subproblem is less complex to design and solve.
3. many problems have a natural hierarchical structure.
4. there may be size limitations on the physical modules.
5. better assignment of subproblem tasks for improved total resource utilization.
6. flexibility and reliability of a decentralized system.
7. lower levels can be used to solve the control problem resulting from estimating the uncertainties in a higher level.

The disadvantages are:

1. complexity of the overall system due to the time dependent interactions of the modules results in changes to the control problem, e.g., the number of variables, the performance index, etc.
2. difficult to analyze the functions of the system, e.g., recognize the state of the system.
3. lack of formal theories to aid in the design and interpretation of the system.

In view of designing an operating system as a controller for the distributed network, the previously mentioned disadvantages would be apparent in any system. Thus, using a hierarchical structure does not compound the difficulties. The ability to have portions of the system running and tested while other subsystems are still being created is far more significant in easing the inherent difficulties of on-line, real-time systems than traditional approaches.

2.3 Application of Control Theory Concepts to Operating Systems

Control theory applications to computer systems have been of a limited nature. Many systems have used performance measures to improve system with or without on-line feedback to the system executive. Systems such as IBM 360/370 often depend upon a "man-in the system" to close the loop of the feedback system. Many memory management schemes use a feedback control, but these applications did not make explicit use of control theory.

Some interesting performance indices have been used to improve system performance with a closed loop, feedback controller. For example, Lowe in [2.5.11] used transfer penalties to measure the amount of information passing between the boundaries of memory hierarchies to improve memory management. The entropy function was used as performance index, e.g., Hellerman in [2.5.12], in an open loop system to measure the amount of work in terms of bits processed by reducing the domain size (memory utilization). Another application, besides memory management, of feedback control in an operating system is in the area of resource utilization, e.g., Flynn in [2.5.13]. These systems are some examples of performance indices other than the usual systems performance criteria of throughput and grade of service.

In a non-operating system environment, which is related to control and command systems, Salton used performance measures to determine the accuracy of document retrieval with respect to a concordance in [2.5.14]. The retrieval system requires people in the system to determine the relevance of the document. Other attempts to improve system performance in terms of reliability, stability, and adaptability have arisen independently of control theory principles.

The first report in the open literature of an attempt to apply control theory concepts to an operating system was by Northouse and Fu [2.5.7 & 2.5.8]. This system applied adaptive control to a job scheduler for an operating system for a real-time, general purpose environment. Specialized on-line systems for ballistic missile defenses, [2.5.9 & 2.5.10], and process control environments, [2.5.15 & 2.5.16], have utilized adaptive and/or self-organizing control systems.

2.3.1 System configuration in Northouse and Fu

The system at NASA for which Northouse and Fu (which offered valuable insight into developing the hierarchical controller) developed the "dynamic scheduler" consisted of two Univac 1106's and four Univac 1108's. One 1106 is used as a front end preprocessor and the other is used for scheduling the four 1108's. The primary goal of the adaptive controller is to maximize throughput by partitioning

the jobs into classes and to assign each job class to one of the 1108's within the constraints of the system.

On arrival to the system each job is preprocessed. Preprocessing consists of task identification from the job control language. By the use of cluster analysis, four job classes were derived from a sample job stream. Scheduling attempts to optimize the throughput of the system as a function of resource allocation, i.e., assigning tape drives and printers to the job class. Within the system, there is no interrupts between program priorities to be processed. Also, job swapping and overlaying are not allowed.

2.3.2 Statement of the optimal control problem considered by Northouse and Fu

The four job clusters, defined a priori, are: (1) medium cpu, large tape files; (2) large jobs; (3) small jobs; (4) medium cpu, small tape jobs. The cluster criteria used to fit a program into a cluster are: programming language used, cpu time requested, actual cpu usage, ratio of requested to used cpu time, Fastran files, tape file, output pages, and the amount of input.

The statement of the optimal control problem for the control system reduces to a linear programming problem as follows. Let x_i, the control variables, represent the number of jobs of class i to be processed during the time interval T, where i=1,2,3,4. Let b_i, the system parameters,

represent the processor weighting. All $b_i=1$ implies that the weighting is equal. The performance index becomes: minimize - $PI = \sum_{i=1}^{4} b_i x_i$, i.e., the throughput of the system is the total number of jobs of each class to be processed within time interval T.

Constraints for the problem are in terms of the measurable quantities (these will be the state variables) of the physical resources, i.e., the number of tape drives used, the printing rate, and the cpu usage for the processors. These were the dominant features produced by cluster analysis. Therefore, if T is the time interval under consideration, the constraints become:

$\Sigma T_i x_i \leq CT$ where T_i represents the mean time of the cpu usage by a job of class i and C is the number of processors;

$\Sigma P_i x_i \leq PT$ where P_i is the mean printing rate (lines/sec) for each job class and P is the mean print rate;

$\Sigma M_i T_i x_i \leq MT$ where M_i represents the mean number of tape drives for each class i and M is the number of tape drives available.

The bounds on these constraints can be modified by reflect the equipment status and the job queue status at the start of the time period considered. Let E_α represent the percent of equipment available and let the Q_α represent queue (the number of jobs remaining to be processed) requirements

where

$$\alpha = \begin{cases} C & \text{runtime} \\ P & \text{print time} \\ M & \text{tape utilization} \end{cases}$$

The modified linear programming problem can be stated as follows:

$$\text{minimize} \quad -PI = b_1x_1+b_2x_2+b_3x_3+b_4x_4$$

$$\text{subject to:} \quad T_1x_1+T_2x_2+T_3x_3+T_4x_4 \leq CT\ E_c - Q_c$$

$$P_1x_1+P_2x_2+P_3x_3+P_4x_4 \leq PT\ E_p - Q_p \quad \text{(eq. 2.2)}$$

$$T_1M_1x_1+T_2M_2x_2+T_3M_3x_3+T_4M_4x_4 \leq MT\ E_m - Q_m$$

$$\text{and } 0 \leq L_i \leq x_i \leq N_i \text{ for } i=1,2,3,4$$

where N_i is the number of jobs in the i^{th} class and L_i is the minimum number of jobs from the i^{th} class to be executed, usually zero.

Since a precise model of the controlled processes is not available, the proposed controller is designed as an adaptive, feedback (closed loop) control system.

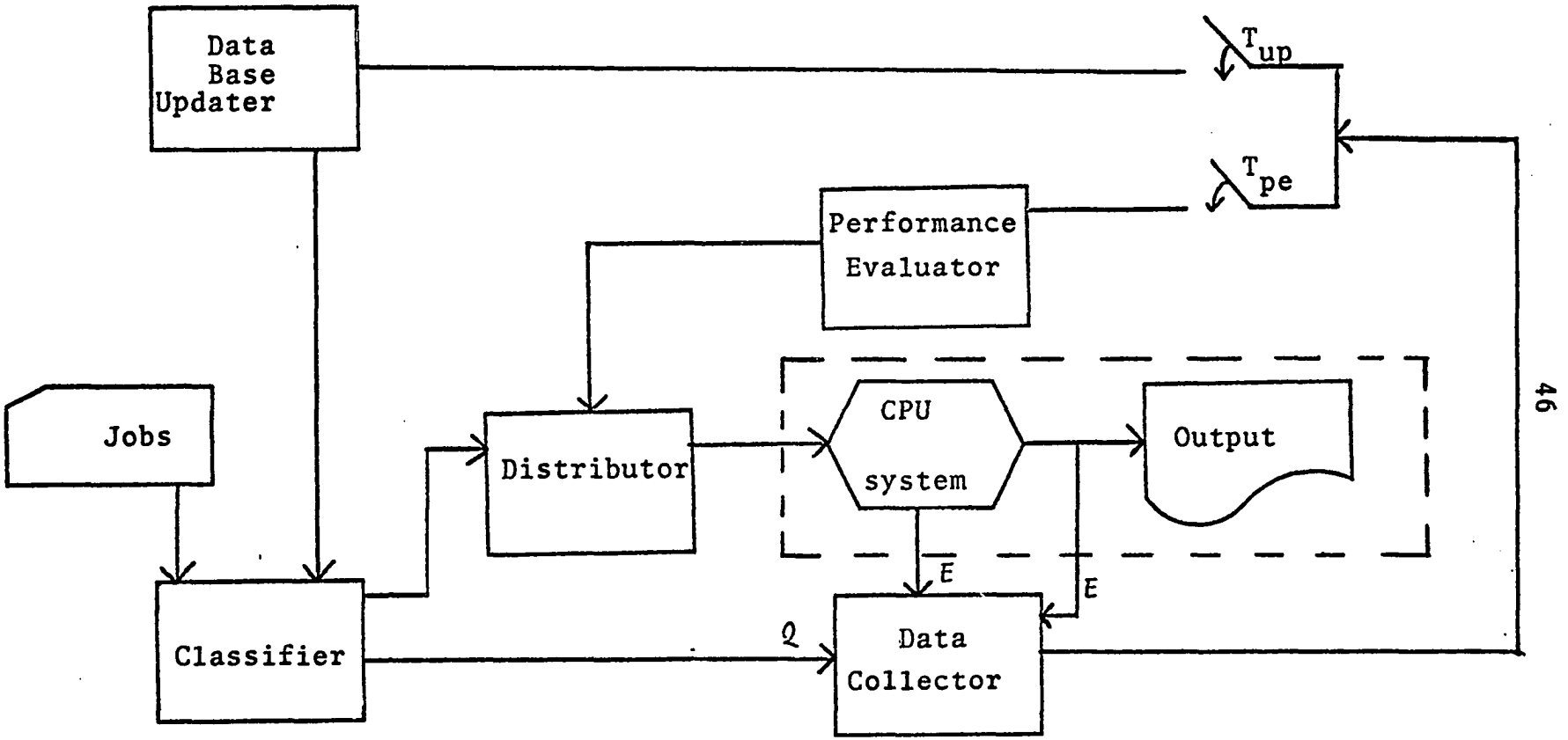
The primary components of this control system are: the classifier, the performance evaluator, and the distributor. As a job enters the system, the classifier uses job stream statistics to evaluate the job control description and determine job class membership. The jobs are then queued by the distributor for execution. The distributor implements the control policy (assigns the number of jobs in each queue to processor), which is the solution to the linear

programming in eq. 2.2. The performance evaluator produces the solution to the linear programming problem at time interval T_{pe} and passes the control policy to the distributor.

As jobs are processed by the computing system, the data collector updates the job statistics by evaluating equipment status and the queue size of the class. When the equipment status changes or queue sizes become too large, this information is passed to the performance evaluator to determine a new control policy. The closed inner loop (i.e., one through Performance Evaluator, Distributor, and Data Collector) in Figure 2.6 represents a performance adaptive controller since the weights, b_i in eq. 2.2, of the job are adjusted to improve the throughput of classes by reflecting changes in the job stream, e.g., to run more small job during the day and more large, IO bound jobs at night.

The data collector also passes information to the data base updater at time instants T_{up}, usually $T_{pe} = T_{up}$, which closes the outer loop in Figure 2.6. The data base updater represents a parameter adaptive controller with learning characteristics by allowing the a priori statistics used by the classifier to be updated. The values of P_i, M_i, and T_i are averaged to represent the job class characteristics and the averages are produced for the a priori job classification criteria.

Figure 2.6 Dynamic Scheduler



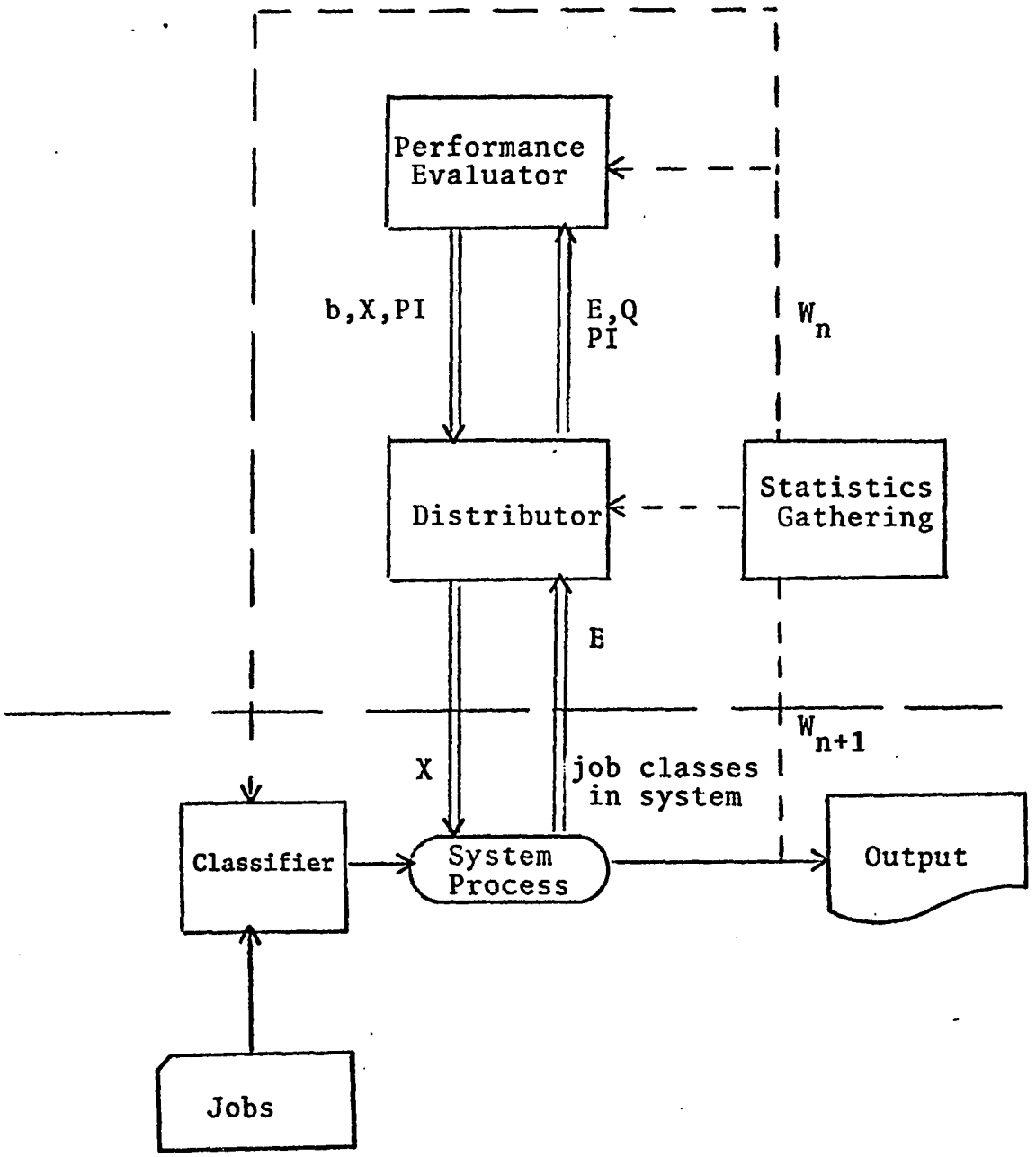
E - equipment status

Q - Queue size

The system of Northouse and Fu satisfies the definitions in Chapter 2 of being controllable and observable. Their system is controllable in the sense that there always exists a control policy (i.e., the assignment vector for the number of jobs to be executed during the time interval). It is observable in the following sense. By observing the output vector, which states the number of jobs of each class processed during the interval, and using the performance index, it may be only statistically possible to estimate the input control policy. If the performance index provides enough descrimination between the number of processors available, then it is possible to determine the equipment status, e.g., if $PI_3 << PI_4$ then with the estimate of the threshold values ($a_3 \leq PI_3 \leq b_3$), then user can determine that 3 processors were available. The system is reliable in the sense that the time frame is fixed in such a way as to prevent the execution of the optimization routine too frequently (so that a "thrashing" type of behavior does not occur). Other advantages of the Northouse and Fu system are: (1) dynamic control rather than static priorities for jobs, (2) simplified scheduling scheme, (3) adaptable to the hardware state of the system.

A reorganization of Northouse and Fu system into one which is hierarchically structured is shown in Figure 2.7. Such a hierarchically organized, multi-level control scheme would have several advantages, as already noted, over the

Figure 2.7 Dynamic Scheduler Represented as a Hierarchical Control System



original Northouse and Fu model due to the increased flexibility and extendibility. In the hierarchical scheme, the lower level is the direct control level that is executed every time frame. The upper level is the adaptive, self-organizing level that is executed only as necessary, at T_{pe} or T_{up}. Statistics are continuously gathered as jobs enter the system, are processed, and leave the system. Statistics reporting to each level and control flow can be performed on basis of the system behavior to reduce the overhead required for the operating system.

An important limitation of the Northouse and Fu scheme is that interactions between programs are neglected. Also, there are no real-time response constraints, such as the IO response time or the sampling rate requirements. Realization of these requirements could have the additional effect upon the system by causing the loss of information due to changes in the system, system overhead demands, and scheduling priorities.

In the next chapter, an example of a two-level control system for a particular distributed network configuration will be presented that will adapt to the state of the system as well as allow for interactions between programs within real-time, on-line response requirements. The system has alternative control techniques with a built in decision mechanism to select the amount of fine tuning required to minimize the bus traffic and to minimize the number of messages lost.

2.5 References

control:

1. Kirk, D. E., Optimal Control Theory, an introduction. Prentice-Hall, 1970.
2. Hvarinen, L., Mathematical Modeling for Industrial Processes. Springer-Verlag Notes on Information and Economic Models, no. 19, 1970.
3. Lewis, J., editorial, IEEE trans. on Automatic Control, Newsletter, December 1970.
4. McGrady, T. P., Stochastic Systems and State Estimation. John Wiley, 1974.

hierarchical control:

5. Mesarovic, M. D., "Multilevel systems and concepts in process control." IEEE (Proc. of the), vol. 58, no. 1, January 1970, p. 111.
6. Schoeffler, J. D., "On-line multilevel systems." Wismer, D. A., ed. Optimization methods for large-scale systems. McGraw-Hill, 1971.

applications:

7. Northouse, R. A., and Fu, K., "Dynamic scheduling of large digital computer systems using adaptive control and clustering techniques." IEEE trans. on Systems, Man, and Cybernetics, vol. smc-3, no. 3, May 1973.

8. Northouse, R. A., PhD dissertation, Purdue, January 1972.

9. Anderson, G. A., "Interconnecting a distributed processor system for avionics." First annual symposium on Computer Architecture, 1973.

10. Jensen, E. D., "A distributed function computer for real-time control." Second annual symposium on Computer Architecture, January 1975.

11. Lowe, T. C., "Analysis of an information system model with transfer penalties." IEEE trans. Comp., vol. c-22, no. 5, May 1973, pp. 469-472.

12. Hellerman, L., "A measure of computational work." IEEE trans. on Comp., vol. c-21, no. 5, May 1972.

13. Flynn, M. J., "Some computer organizations and their effectiveness." IEEE trans. on Comp., vol. c-21, no. 9, September 1972, pp. 948-960.

14. Salton, G., Automatic Information Organization and Retrieval. McGraw-Hill, 1968.

15. Lee, T. H., Adams, G. E., and Gaines, W. M., Computer Process Control: Modeling and Optimization. Wiley, 1968.

16. Savas, E. S., Computer Control of Industrial Processes. McGraw-Hill, 1965.

17. Dijkstra, E. W., 'The structure of "THE" multi-programming system.' CACM, vol. 11, no. 5, May 1968, pp. 341-345.

CHAPTER 3

A TWO-LEVEL HIERARCHICAL CONTROL FOR A DISTRIBUTED NETWORK OF COMPUTERS

The objective of this chapter is to apply control theory concepts to the operating system for a distributed network of computers. For that purpose a mathematical representation must be formulated for the system model. If an optimization model, as defined in Chapter 2, is chosen for this purpose, then the performance criteria must be selected and the physical system constraints must be defined.

The need for a optimization model arises as follows: in a distributed network of computers which are connected by a common bus, the bus often is a major bottleneck. Besides, for two programs residing in two different processing elements to communicate, a certain amount of message-transmission-through-the-bus overhead must be paid on both ends. (As previously mentioned, a processing element is an autonomous computer often a mini- or microprocessor with its own memory, processor, and network interface devices.) To improve the system performance, the goal in this system is to minimize the bus traffic by using a control policy to determine those

cooperating processes that can be assigned to the same processing element. Therefore, it is logical to choose the distribution of the programs to be executed by one of the processing elements, which are the control variables in this system, as the assignment variables.

As already mentioned in Chapter 2, there can be only one performance index in a single level control system. Thus, one of the reasons for selecting a hierarchical representation of the operating system is to include additional performance indices. The number of missed messages due to processor scheduling was selected as such a lower level performance index.

As previously defined, the input-output variables for the control system are expressed in terms of the messages that are input to or created by the set of programs. The propagation of the messages is determined by the IO device acceptance rates and response requirements. The system parameters (a subset of the state variables) includes the following factors: the number of processors, the amount of memory in each processor, the processor status, the program-message descriptors, current loading system locating information, etc.

In this chapter, a two-level control scheme is described along with the assumptions concerning the hardware configuration. The control level that determines the optimal assignment of programs to processors is referred to as the

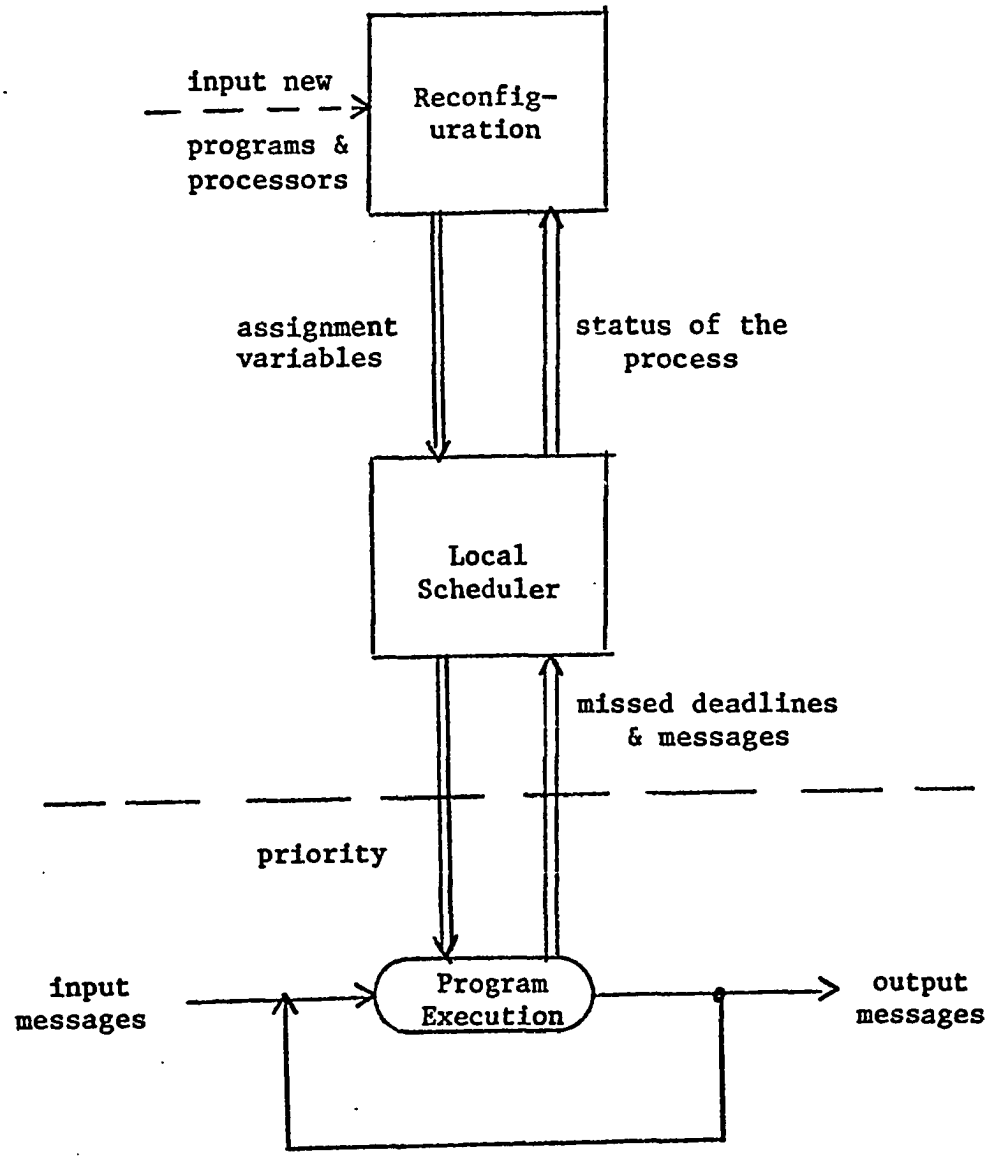
reconfiguration level. Since this level may cause unnecessary system overhead if executed too often, it has been designated as the higher (or adaptive) control level, which is to be evoked only after the number of processing elements change or when additional programs enter the system. The lower control level, or direct control, is represented by the local schedulers of processing elements. Such a scheduler tries to schedule the programs residing in the processing element under its control so as to minimize the information losses due to missing the deadlines of the over-the-bus transmitted messages. This problem arises mainly due to the distribution of cooperating processes over a set of processing elements.

Figure 3.1 shows the overall structure of such a two-level control scheme for computing systems. Algorithms for reconfiguration are considered in Chapter 4. A simulation of the hierarchical control model is presented in Chapter 5.

3.1 Hardware and software assumptions

The purpose of this section is to define the model of a computing element which later is to be used as a reference point for developing control-theoretic concepts for computer executive design. Given a set of programs, the IO device acceptance rates and the response requirement rates determine an overall message rate and program execution rhythm. (For simplicity, assume that the schedules of all programs are

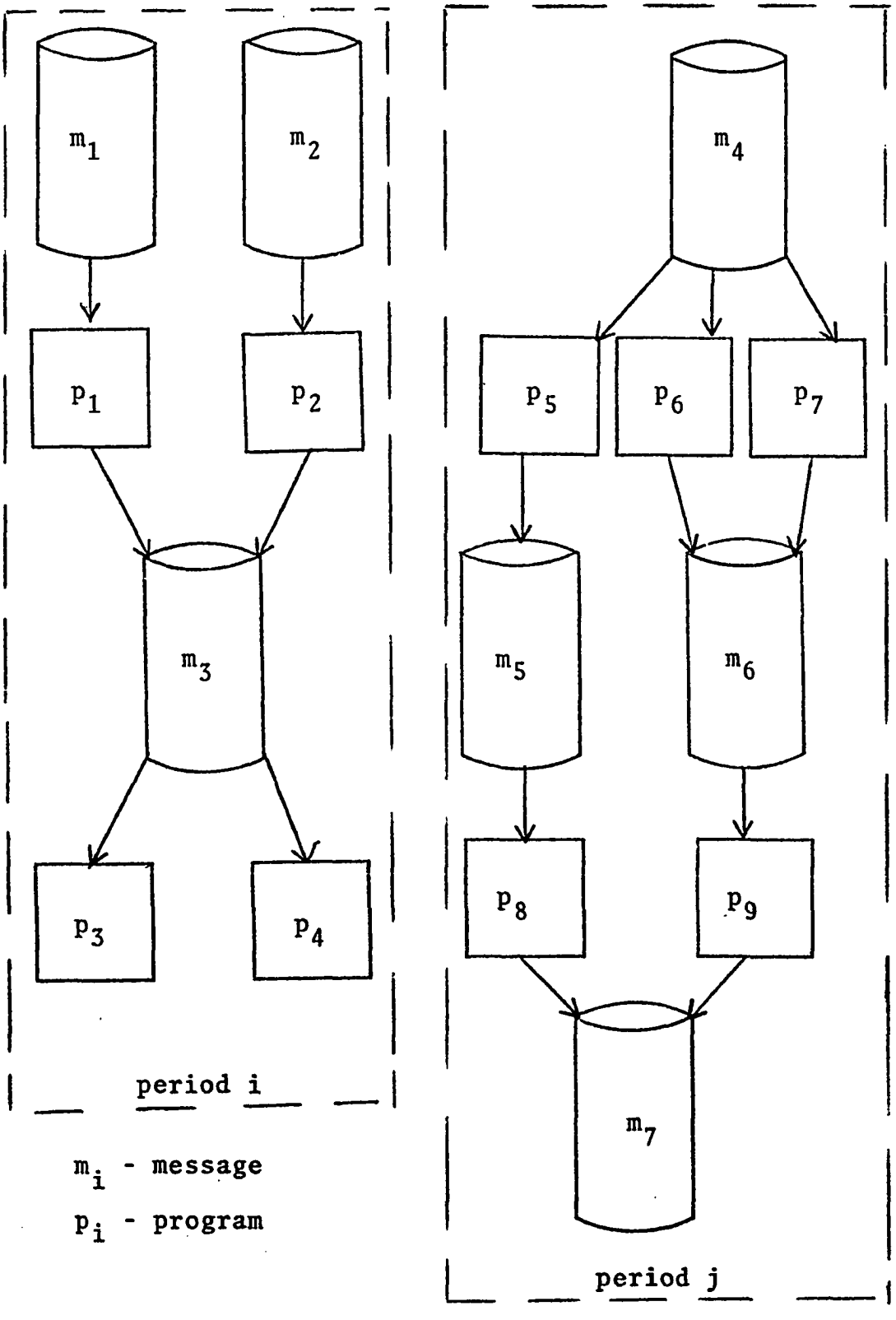
Figure 3.1 A Two-level Hierarchical Controller



cyclic.) These messages can be viewed as the members of data sets within a process schema, as shown in Figure 3.2. Within a process schema [3.5.6] the message rates dictate program execution rates, which will be called program frequencies. In the example in Figure 1.2 in Chapter 1, the messages were to be generated once in every 400 time units which required that the programs be executed at the same rate. However, in the present model, all the message rates do not have to be the same, e.g., in Figure 3.2 messages m_1 through m_3 occur at a rate of once every i time intervals and m_6 through m_9 occur at a rate of once every j time intervals. This time interval (called the standard time frame) is a fixed, but finite time interval that is determined in such a way as to be able to handle the program run-time necessary for the processing the heaviest message traffic. The program period is defined as the deadline by which the program must complete execution and is measured in terms of the number of standard time frames, e.g., program p_1 in Figure 3.2 must complete execution within every i standard time frames since the period of program p_1 is i.

Consider the example for a set of programs A, B, and C which have periods 1, 2, and 4, respectively. Let the standard time frame be 16 time units. Program A must be executed before the completion of every standard time frame, i.e., every 16 time units. Program B (C) must complete

Figure 3.2 Process Schema



execution before the end of every 32 time units (64) time units. Usually the smallest period is one, but the model can accommodate fractional periods if necessary. Note that the period as well as the standard time frame are usually expressed as powers of 2.

This frequency method is used in order to control cooperating processes, rather than using a complex method involving interrupts and semaphores. The cooperating processes in the process schema are those that share the same data, e.g., in Figure 3.2 m_3 is to be utilized by both programs p_3 and p_4 before the occurrence of another message, say m_6.

The configuration problem constitutes the assignment of interacting programs along with the required data sets which can reduce the utilization of the common bus. However, the constraints of the processor due to the amount of memory available and the limited amount of run-time within the standard time frame can prohibit the assignment of all cooperating processes to the same network element. Therefore, the controller that produces the reconfiguration is a performance-adaptive, self-organizing process. Messages can be "lost" during program execution due to the assignment of a program that cannot be scheduled for processing at a rate equal to or greater than the message rate. Double buffering can be used to extend the length of "grace" time that the message is available to the processor.

To keep the model simple, program priority interrupts are not used. The only interrupt that is processed by the system is the synchronizing signal which resets the clock within each processing element at the start of every standard time frame.

The protocol for the messages is the priority scheme used to determine the precedence relation for transmitting messages on the bus. In this system, the message with the lowest identification number will have the highest priority on the bus. The identification number is determined by the ratio of the duration of the message to the period of the message and these numbers are ranked in increasing order. Protocol and bus arbitration schemes are discussed in [3.5.3], [3.5.4], and [3.5.5].

3.2 Reconfiguration

The optimizing level of control (reconfiguration) solves a quadratic binary programming problem with side constraints which is developed in this section. Reconfiguration is the process of determining an optimal assignment of programs and data bases to the processors. (Determination of the initial configuration may be considered as a process construction task in which the techniques very similar to those discussed here can be applied.)

To state the performance index of the total amount of process interaction within a computing element, a quantative

measure of message and program interaction must be introduced. Let x_{pj} represent the assignment variable, i.e., control variable, of the status of program p within processor j such that $x_{pj}=1$ if the program is assigned and $x_{pj}=0$ otherwise. There are n programs to be assigned to k processors. Let c_{pq} be the amount of interaction between programs p and q as measured in terms of time. Let t_i be the amount of time needed to transmit message i on the bus, which is approximately proportional to the message length in bits, and let m_{pi} be the status of message i with respect to program p, i.e., $m_{pi}=1$ if program p uses message i and $m_{pi}=0$ otherwise. Then, the amount of interaction between programs p and q for u messages is

$$c_{pq}=\sum_{i}^{u} t_i m_{pi} m_{qi} \ .$$

The total amount of interprocessing communication time saved for the program assignment to processing element j is given by

$$\sum_{p<q} c_{pq} x_{pj} x_{qj} \ . \qquad \text{(eq. 3.1)}$$

The c_{pp} could be used to represent the program loadings since $x_{pi} \cdot x_{pi}=x_{pi}$, but they are not used in this model. (To be more precise, eq. 3.1 gives the amount of bus traffic, in terms of time, eliminated by placing programs p and q into the same processing element.) Therefore, the performance index is the sum of eq. 3.1 over j for all k processing elements.

Constraints on the processor j are:

$$\sum_p s_p x_{pj} \leq M_j$$

$$\sum_p r_p x_{pj} \leq T,$$

where s_p is the number of pages of storage required by program p, M_j is the total number of pages in the memory of processing element j, r_p is the run-time for program p, and T is the length of the standard time frame.

Therefore, the optimal control problem can be stated as follows:

maximize PI, where $PI=\sum_j^k \sum_{p<q} c_{pq} x_{pj} x_{qj}$

subject to: $\sum_p s_p x_{pj} \leq M_j$ (eq. 3.2)

for j=1,2,...,k processors

$\sum_p r_p x_{pj} \leq T$

for p=1,2,...,n programs

and $\sum_j x_{pj}=1$

where $x_{pj}=0$ or 1.

Since all of the coefficients and bounds in eq. 3.2 are positive integers, the problem is a quadratic zero-one programming problem. (In the literature, the quadratic assignment problems are of the form of eq. 3.2 but do not have the inequality "side-constraints" stated above in terms

of memory and run-time limitations.

Thus, on this control level, applying the definitions of Chapter 2, the control policy is represented by the solution to the quadratic binary programming problem. This policy is to be passed on to the lower level, the schedulers, of the individual processing elements. Control parameters of this performance-adaptive, self-organizing level are the coefficients, bounds, and program-message descriptors needed to form eq. 3.2. The list of all available processing elements and a description of each also belong to the control parameters. The input vector for this control level comes from the lower level which reports the number of changes in the processor status at the beginning of every standard time frame. Other input variables to the higher level are external to the computer system; such inputs can change the number of programs and processors in the computer system.

It is very difficult to produce an optimal solution to eq. 3.2 within the constraints of an on-line, real-time system, i.e., in real-time. These difficulties are caused by the problem size. For example, there are n*k control (assignment) variables with a maximum of 2^{n*k} possible solutions, and with 2*k+n constraints.

3.3 Local scheduler, or direct controller

Scheduling, or direct control, implements the control

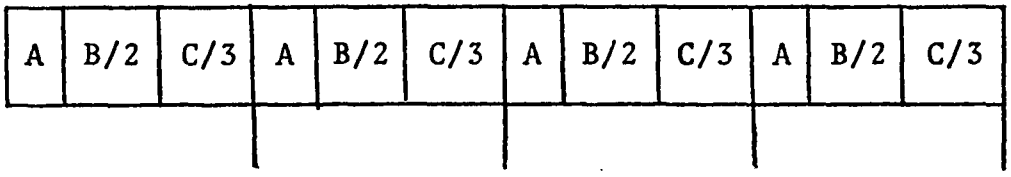
policy from the self-organizing level. Scheduling has been extensively studied in the literature; a good bibliography on scheduling is found in [3.5.11]. Two methods for the local scheduler are considered here which is to be implemented in each computing element.

Using the definitions of state variables from Chapter 2, the control inputs to this control level are the program assignment vector (control policy) for the programs to be executed by each processor. The control parameters on this control level are the program run-time, program periodicity, and the specification of the length of a standard time frame. The control policy to be produced is the schedule for executing the programs and the performance index is the number of missed message bits or words. (Messages could also be weighted according to their importance.)

Consider the following heuristic method for scheduling periodic tasks [3.5.9] which guarantees real-time deadlines and is explained by means of an example. The advantages of using this heuristic algorithm are as follows: the task list can be obtained with a minimum local system overhead; the number of interrupts needed to execute the task list is minimized (also saving overhead in storing program status), and the fail soft capability is provided by completing the most imminent tasks first.

3.3.1 Example 1 for processor scheduling

Let T be the length of a standard time frame. The period of the program is the length of time measured in terms of the number of time frames before the occurrence of the next message which is to be input or output by the program. Consider the tasks A, B, and C with periods of T, 2T, and 3T, respectively. A schedule to meet these deadlines would be to execute A every time frame, one-half of B every time frame and one-third of C every time frame as shown below:

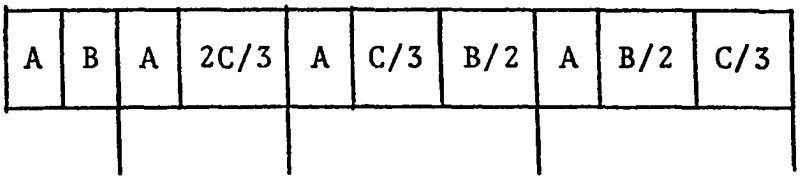


Deadlines:	T	2T	3T	4T
Programs meeting the deadlines	A	A B	A C	A B

3.3.2 Example 2 for processor scheduling

To reduce the number of interrupts, Jordan's rule is used to consolidate as much of the program's execution in one time frame as it is possible without violating the program deadlines [3.6.9] and [3.6.12].

Therefore, the above schedule would become as follows:



Deadlines:	T	2T	3T	4T
Programs meeting the deadlines	A	A B	A C	A B

3.3.3 Limitation on bus traffic model

In the model used in this thesis, the program-message transition table for the IO is determined at process construction time and remains fixed, except in the case when exogenous messages are introduced into the system. (Aperiodic messages can be considered either at a higher priority or a lower priority than the periodic messages.) The consideration of aperiodic data could cause further loss of information in terms of missed messages. When so many processors fail that all of the programs cannot be executed within the time frame, then some control policy must exist to select the most important functions.

3.4 Summary

In the Northouse and Fu model, presented in Section 2.3, the main goal was to study the classifier and the distributor of the dynamic scheduler and not to study the

representation of an operating system as a controller. This chapter was concerned with outlining the approach to define a controller for the distributed network of computers which used a hierarchical structure.

In such a network, the process inputs as well as outputs of this network of computers constitute a set of programs and messages. This system is observable in the sense that the outputs can be enumerated as they are produced. The system is controllable, again in a weaker sense, in that some control policy does exist for each time frame even though all of the programs may not be executed or some of the messages are lost.

The "information" vectors are required by the levels of the hierarchical control to produce the control polices for the computer network, i.e., processing elements. As noted above, the process inputs and outputs are the programs and I/O messages. The lower level (scheduler) receives the assignment vector (the control policy) from the higher level (the reconfiguration level). The control policy which is produced on the lower level is the schedule for the programs assigned to that processing. The control vector that is passed to the higher (or reconfiguration) level is the status vector for the processing elements.

The information (in vector Z) needed by the state of the system to produce the reconfiguration must specify the length of the standard time frame, the processing element

status, their size and speed limitations, the program-message-interaction descriptors, and the message times for the bus transmission (e.g., the tables used in the example in Chapter 1). The information on the system needed to produce the schedule is the program execution rates and periodicity and the program-message-interaction descriptors. The noise is introduced into the system by processing element failures, aperiodic message traffic, and by the inability to keep up with the message traffic rate. The performance indices are the savings in the amount of bus traffic (reconfiguration) and the number of missed messages (scheduler) for the standard time frame.

In the next chapter, methods for producing the control policy for the reconfiguration level. It was only after this investigation that the controllability on the reconfiguration could be determined.

3.5 References

modular systems:

1. Dijkstra, E. W., 'The structure of "THE" multi-programming system.' CACM, vol. 11, no. 5, May 1968, pp. 341-345.
2. Jansen, E. D., "A distributed function computer for real-time control." Second Annual Symposium on Computer Architecture, January 1975, pp. 176-180.

bus communication systems:

3. Chen, R. C., Bus Communication System. PhD dissertation, Carnegie-Mellon, January 1974.
4. Thurber, K. J., "A systematic approach to the design of digital busing structures." Proc. AFIPS conf. vol. 41 (FJCC '72), pp. 719-740.
5. Nisnevich, L. and Strasbourger, E., "Decentralized Priority Control in Data Communications." Second Annual Symposium on Computer Architecture, January 1975, pp. 1-6.

process schemata:

6. Horning, J. J. and Randell, B., "Process Structuring." ACM Computing Surveys, vol. 5, no. 1, March 1973.

7. Sorenson, P. G., "Interprocessor Communication in Real-time Systems." IEEE Symposium on Computer Software Reliability, January 1974, pp. 1-7.
8. Johnson, D., "A process-oriented model of resource demands in large multiprocessing computer utilities." PhD dissertation, University of Texas at Austin, 1972, pp. 26-44.

scheduling:

9. Jordan, J. W., "Task scheduling for a real-time multiprocessor." NASA TN B-5786, May 1970.
10. Ramamoorthy, C. V., Chandy, K. M., and Gonzalez Jr., M. J., "Optimal scheduling strategies in a multi-processor System." IEEE trans. on Computers, vol. c-22, no. 2., February 1972, pp. 137-146.
11. Coffman, E. G. and Denning, P. J., Operating Systems Theory. Prentice-Hall, 1973.
12. Thurber, K. J. and Jack, L. A., "Time Driven Scheduling." COMPCPN 1974, pp. 181-185.

CHAPTER 4

RECONFIGURATION TECHNIQUES

In the model of the hierarchical control system introduced in the last chapter, the state of the system of processors is transmitted periodically to the reconfiguration control level. The reconfiguration of the system, reassigning programs to processors, occurs after the number of active processors changes due to a failure or a recovery. Changes in the number of active programs or in the data base used by programs may also trigger reconfiguration. The assignment algorithm used to perform the reconfiguration must not only run in a real-time environments but also minimize the amount of inter-processing element communication.

There are several methods to achieve reconfiguration, but two methods were selected primarily due to the memory restrictions placed on the processing element size. Since these methods were not investigated in the current literature for precisely the same problem statement as eq. 3.2, the methods were investigated in this chapter to determine their suitability for performing the reconfiguration with the additional "side constraints", i.e., the inequality

constraints of eq. 3.2. One method (quadratic zero-one programming) produces an optimal solution. If this method does not satisfy the real-time processing constraints, then a sub-optimal method (cluster analysis) is necessary. Both methods, the quadratic zero-one programming and cluster analysis, have several "variants" to improve their performance which are also included here.

Non-hierarchical cluster analysis is a heuristic method of assigning items to a fixed number of clusters so that "similar" items will fall in the same cluster. To use cluster analysis to assign programs to a processing element based on the amount of input/output shared among the programs, bounds must be placed on the cluster so that all of the programs assigned to the cluster will execute within the time required and will fit in the processor memory. Since the results of cluster analysis are sensitive to the initialization of the cluster centroid, the distance function used, and the iteration scheme, this chapter compares variations in the non-hierarchical cluster analysis under timing and memory constraints.

Normally, cluster analysis methods can produce at most a sub-optimal solution to the problem of assigning programs to processing elements. As noted in the first paragraph, optimization of computing network configuration means minimization of the inter-processing element communication within the existing time and storage constraints. Quadratic

zero-one programming can at least theoretically be used to compute an optimal solution, although it was found to have excessive run-time requirements. Therefore, the quadratic binary programming algorithm is used to compare the results of the cluster analysis algorithms. It is possible to use quadratic zero-one programming at process construction time to determine the initial configuration of the processes or to pre-compute and save the assignments as the control policies to be subsequently used for reconfiguration in real-time.

4.1 Quadratic Binary Programming with Side-constraints

Few methods exist for producing an optimal solution for large scale quadratic binary programming problems as stated in eq. 3.2 for problems containing as few as 40 assignment variables. In this section, methods are reviewed to improve the performance of the quadratic binary programming algorithm with side-constraints. Early attempts only considered problems with fewer than 15 assignment variables and with no side-constraints. Increasing the number of variables increased the execution time exponentially. The cutting plane and the implicit partial enumeration algorithms are considered for producing an optimal solution; [4.6.1], [4.6.2], [4.6.3], [4.6.10], [4.6.11], [4.6.13], [4.6.14], and [4.6.15] do not consider the side-constraints. These constraints can improve the bounding in the tree search

method, but require additional storage and computation time.

For the purpose of this study, the order of the quadratic binary programming problem is defined to be the number of assignment variables. For n programs to be stored in k processors, the order of the quadratic programming problem is nk, where usually n>>k. There are 2k constraints on the processors for the side-constraints. The number of terms in the objective function is n(n-1)k/2. In this section the objective function is stated as a maximization problem in order not to have to introduce a substitution of variables $z_{pi}=1-x_{pi}$ which would add linear terms to the objective function.

To solve this problem (eq. 3.2) the discussion in [4.6.2], [4.6.10], and [4.6.15] were limited to the order of not greater than 15, with the only constraint being to restrict the assignment variables to binary values and to guarantee the assignment of all of the items to at most one location. In most real situations, sub-optimal solutions techniques are most frequently found in the literature and produce acceptable results. Some of these techniques, such as cluster analysis as in [4.6.19] and [4.6.2], may fail to produce assignments in tightly constrained cases, e.g., where approximately 90% of the system resources are required to execute the set of programs.

Tableau methods (see [4.6.2] and [4.6.16]) after a cursory study on each algorithm may require too much storage

space, e.g., (3nk+2k+3)(nk+2k+2) for the tableau in [4.6.16]. Also, methods (using branch and bound techniques) may have to store the tableaus on auxiliary storage devices. This increases the execution time due to retrieval of the tableaus during "backtracking". Other classical approaches have required too much execution time, e.g., Lawler or Gilmore used 30 sec. for a problem with only 15 variables [4.6.17]. In general, algorithms that have been implemented have been too specialized to be useable except in limited cases [4.6.18]. A "gradient" approach using difference techniques in [4.6.9] is impractical to generate the additional constraints on the assignment variables for a problem of order greater than 5; this would require a method for solving Karnaugh map problems [4.6.21] that would take longer than the original problem. Most of these methods have only been applied using hand calculations and have not been considered for large scale problems of order greater than 40.

The only algorithm that has been implemented for large scale problems was found to be the implicit partial enumeration algorithm, [4.6.5] or [4.6.17], and restated in subsection 4.1.3. For a problem of the order of nk, the maximum number of nodes that could be generated in the tree search is 2^{nk} if a total enumeration occurs. Even if only 50% of the nodes must be investigated, then 2^{nk-1} nodes must still be generated. Acceleration techniques are considered

in the next section to reduce the number of nodes that must be investigated.

4.1.1 Acceleration techniques

To apply the implicit partial enumeration algorithm to the quadratic binary programming problem the objective function in eq. 3.2 must be linearized, [4.6.8] and [4.6.17]. Therefore, let $y_{pqj}=x_{pj}\cdot x_{qj}$. The new constraints needed to guarantee the appropriate properties are:

$$x_{pj} + x_{qj} \geq 2y_{pqj} \qquad \text{(eq. 4.1)}$$

$$\text{and} \quad x_{pj} - x_{qj} - 1 \leq y_{pqj} \qquad \text{for } p, q \text{ where } p,q=1,2\ldots,n \text{ and } j=1,2,\ldots,k.$$

Adding these constraints to a tableau would increase the size of storage needed by $n(n-1)k$. In the tree search method such as in the implicit partial enumeration, the condition stated in eq. 4.1 can be tested without explicitly storing the large zero-one sparse matrix. The implementation strategy is discussed in subsection 4.1.4.

Surrogate constraints were suggested by Glover (in [4.6.8]) to accelerate algorithm execution time. These new constraints are formed by taking linear combinations of the problem constraints and augmenting the resulting constraint to the initial set. In this problem, two surrogate constraints are formed by summing each of the inequality constraints of eq. 3.2 on j. These constraints stated as

follows:

$$\sum_j \sum_p s_p x_{pj} \leq \sum_j M_j$$

$$\sum_j \sum_p r_p x_{pj} \leq kT \qquad \text{(eq. 4.2)}$$

and are augmented to the original 2k constraints. In systems application that have the same units of measure for the coefficients of the constraints, the problem can be reduced to a network problem using a single surrogate constraint. The inclusion of the surrogate constraints in eq. 4.2 reduced the tree search, i.e., improved the bounding by elimination of nodes, by 75% for the problem in Chapter 1 which was of order 18, see Table 4.1.

Laughhunn, in [4.6.12], proved that the optimality and feasibility conditions used in the Balas zero-one linear programming problem in [4.6.5] and [4.6.17] hold for the quadratic case. These conditions are exclusion tests that depend only upon the constraints and can be applied to the quadratic binary programming problem, since the constraints are also linear. Testing each constraint may increase execution time without yielding an improvement in the bounding in the tree search algorithm. A single exclusion test (called feasibility) that was adopted for the assignment problem discussed in this thesis is:

$$\text{if } \sum_{p \in F} a_p x_{pj} + |a_q| > t_j, \text{ then } x_{pj} = 0 \text{ since all } a_p \geq 0$$

where F is the set of unassigned programs, a=s (or r), and t is the amount of memory (or execution time) remaining in

TABLE 4.1 Comparison of Methods

Order	Effective Number of Constraints	Solution Method	370/150 run-time	Solution Type	Remarks
15	10	Lawler or Gilmore	30 sec*	optimal	Classical approach
18	25	QUAD	4 sec	optimal	Example in Chapter 1. Run-time = 20 sec. without surrogate constraints.
40	180+	QUAD	44 min	optimal	Cluster analysis able to determine a suboptimal solution using only one iteration. (Objective function = 152 in .01 sec.) QUAD objective function = 159 99% of resources utilized.
250	1000+	QUAD	2 min 51 sec	sub-optimal	1 solution generated without the exclusion test. 4 solutions generated (in the same amount time) using the exclusion test. Cluster analysis produced solution in .8 sec (see Table 4.4)

* Adjusted from IBM 7090 run-times by multiplying by a factor of 1/10.

the processing element j after each program assignment. Applying the single exclusion test, i.e., for feasibility, increased the number of partial solutions that the algorithm was able to generate in the same amount of time, see Table 4.1.

Another method used to eliminate nodes is to determine the best improvement of the objective function, referred to here as lower bounding. Lower bounding is not used in this application since all of the coefficients are positive, resulting in the lower bound being zero in each case, i.e., the lower bound for the value of the objective function does not increase over the last partial solution, because the program may have to be assigned to the processing element even though there may not be a reduction in the inter-processing element communication. The amount of search time to find the maximum contribution to the value of the objective function requires j computations of the objective function with j(j-1)k terms for j unassigned programs. Since the number of constraints is small compared to the size of the objective function, many nodes can be investigated within the time needed to find the "best" assignment of a program.

4.1.2 Implicit partial enumeration algorithm

The implicit partial enumeration algorithm was used to find the optimal solution to the quadratic zero-one

programming problem (eq. 3.2) that was stated in the previous chapter. The approach used in this study is: (1) linearize the objective function, (2) augment surrogate constraints, (3) apply the exclusion test, (4) implicitly test the constraints to guarantee the zero-one properties, and then (5) apply an implicit partial enumeration algorithm [4.6.17]. This approach does not require additional memory for the constraints that arise due to the linearization.

The algorithm in Garfinkel and Nemhauser [4.6.4] was implemented using dynamic storage allocation in the QUAD procedure in appendix A. This brought about a restatement of the algorithm using the concepts of a controlled stack of storage whose size can be changed when new elements are inserted onto (or deleted from) the stack. (This feature aids the portability of the procedure by not having to adjust the dimension declarations for a new configuration.)

The implicit partial enumeration method produces a binary tree, Figure 4.1, but only the current path needs to be in storage an any one time, e.g., path abcdef in Figure 4.1. This path will be represented by the tree stack, Figure 4.2, which is a controlled storage area with each node of the stack being an array whose components are the x_{ik} assignment variables with:

$$x_{ik} = \begin{cases} 0, \text{ if available for assignment} \\ 1, \text{ if assigned at the value 0} \\ 2, \text{ if assigned at the value 1} \end{cases}$$

Figure 4.1 Binary Tree
(path≡abcdef)

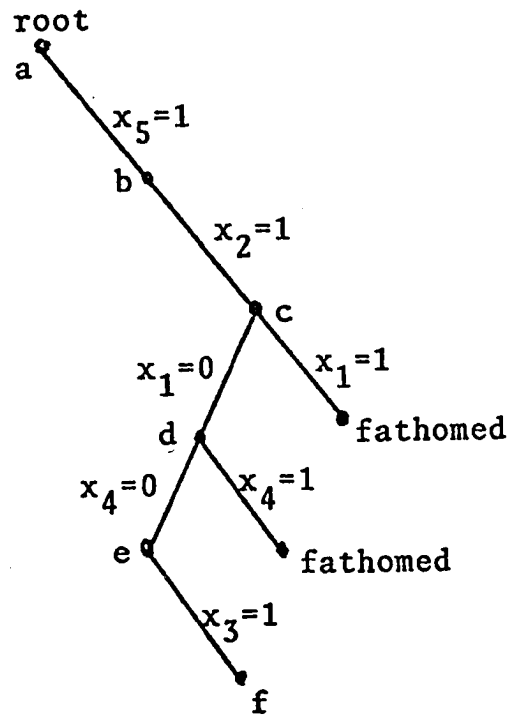


Figure 4.2 Tree Stack
(path abcdef)

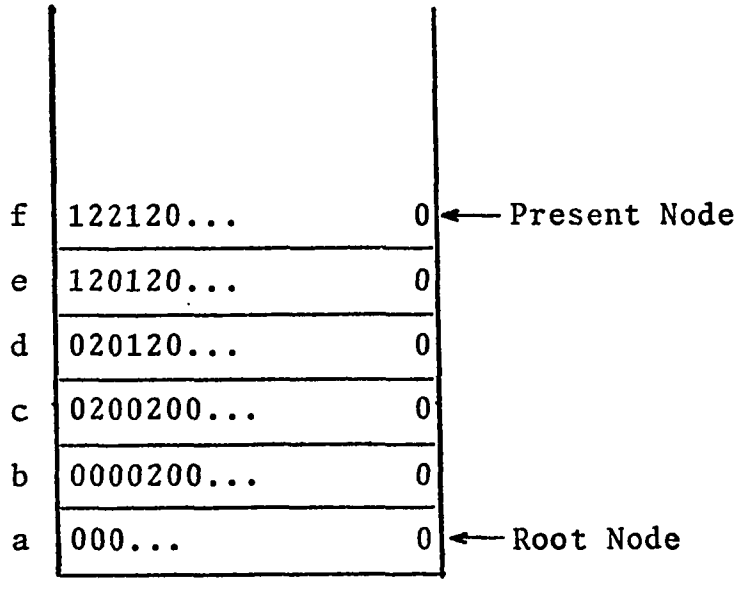
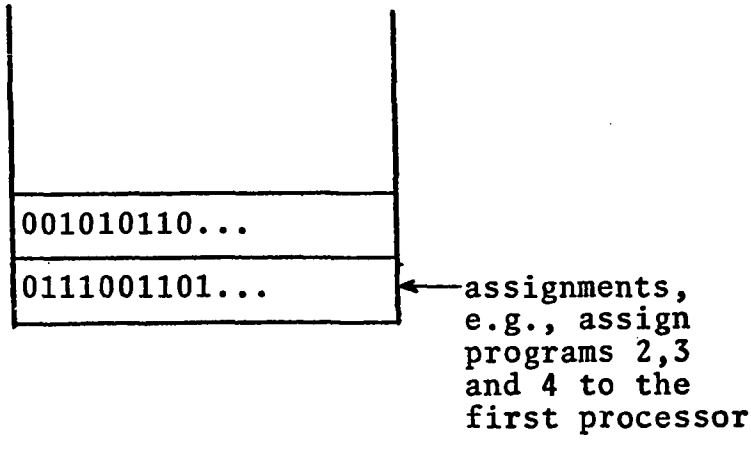


Figure 4.3 Solution Stack



The solution stack, shown in Figure 4.3, will be a similar controlled storage area used to save the nodes which produce the maximum value of the objective function during the execution of the algorithm. By using controlled storage, only the last item on the stack can be accessed by the algorithm. To add a node to either stack, first the space must be obtained by an allocate instruction. Releasing the space to access subsequent nodes is accomplished by free commands. Figure 4.4 shows the contents of the tree stack before x_3=1 is to be augmented to the current partial solution; Figure 4.5 shows the stack contents after the node has been added. To backtrack, the last variable set to one must be replaced by its value set at zero, Figure 4.6. When there is no different between the last two nodes on the tree, the last node that was allocated will be freed before proceeding with the backtrack (Figure 4.7).

4.1.3 Algorithm statement

Quadratic zero-one programming algorithm with linear constraints and positive coefficients.

1. initialization: put zero on the tree stack for the root node and set MAX=0.
2. partition: find index p such that $a_{pi} \leq t_i$ for all i, where p is an unassigned item. If p does not exist then go to fathom. Create the node and place it on the tree stack, i.e., allocate, and set the present node equal to the last node but augmented

Figure 4.4
before x_3 is augmented
to the tree stack

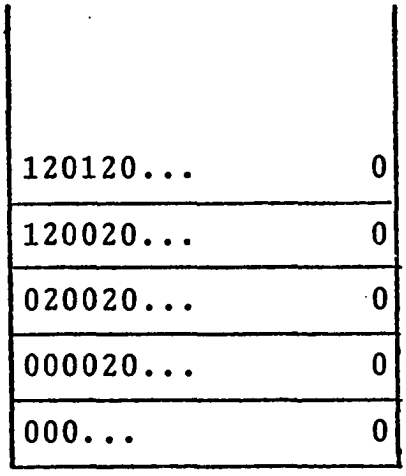


Figure 4.5
x_3 = 1 augmented
to the tree stack

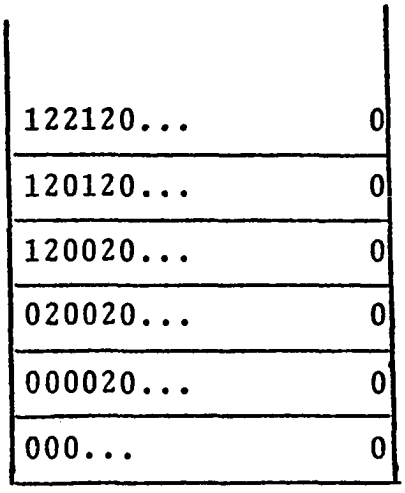


Figure 4.6
x_3 = 0 after the backtrack
on the tree stack

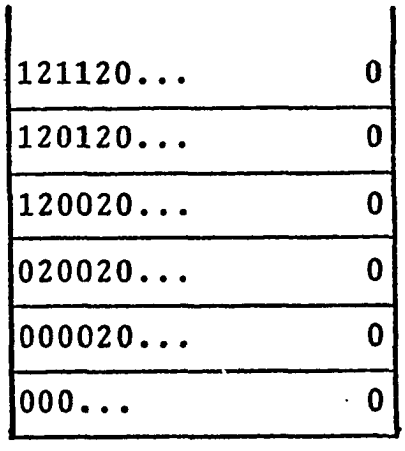
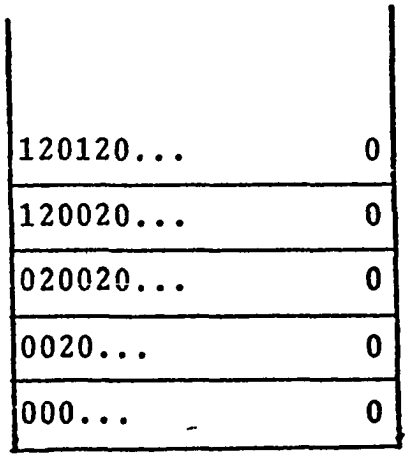


Figure 4.7
free the last node
on the tree stack



by the p^{th} index. Go to calculate.

3. calculate:
   a. compute new bounds, $t_i=t_i-a_{pi}$ for all i.
   b. does the present node represent a solution, i.e., are all assignments made and an item is not assigned to more than one cell? If an item is assigned more than once, then go to fathom. If the current node does not maximize the objective function, then go to partition.
   c. if the current value of the objective function is equal to the MAX value then insert the present node on the solution stack. If the objective function value is greater than the MAX value then the solution stack must be cleared before inserting the present node and changing MAX to the current value of the objective function. Go to fathom.

4. fathom: if the present node is the root, then go to terminate. Otherwise, save the present node and free the space allocated to it. Go to compare.

5. compare: compare the present node and the previous node. If the difference between the components is 2 for index p, then set $x_{pi}=1$ and recompute the bounds, $t_i=t_i+a_{pi}$ for all i bounds. Go to partition. If there is no difference in the positive assignments, then go to fathom (no p index was found)--this performs the backtrack.

6. terminate: print MAX and the solution stack.

The storage requirement for this particular implementation of the algorithm is small, especially considering that auxiliary work area on the disk is not needed. The maximum number of nodes that could be generated for the tree stack is

2^{nk} nodes (if there is no bounding) and for the solution stack is 2^{nk} nodes (if every node is a solution) where the node size is 8nk bits and n is the number of programs and k the number of processors. The maximum depth of the tree stack is nk+1 and the number of coefficients, a_{ij} with $a_{i1}=s_i$ and $a_{i2}=r_i$ for i=1,2,...,n, for the linear constraints is 2n. The $\Sigma x_{ik}=1$ case is tested separately and not stored as a constraint. Even if the processors are identical there still must be 2k+2 constraint bounds, t_i, for a work area. The number of elements in the objective function, c_{ij} where i<j, is stored in a linear array of dimension n(n-1)/2 and is also a controlled storage area.

Storage requirements for the algorithm are further reduced by making use of logical tests to guarantee the binary conditions. Two types of constraints are identified here, explicit and implicit. The explicit constraints are those that require an array for storing the coefficients, e.g., constraints on memory and runtime. The implicit constraints are those that can be determined from the assignment vector by using the logical operations, e.g., the constraint that a program can only reside in a single processor in eq. 4.1. The resulting reduction in memory requirements for the constraints is $2n^2k-2nk-nk=2n^2k-3nk$ bytes. In this problem, the coefficients of the constraints did not need k copies of the constraint coefficients which further reduced the memory requirement by (2k+1)n words of

storage for the 2(k+1) constraints. For n=50 programs and k=5 processors, excluding implicit constraints saves 24250 bytes of memory and excluding k copies of the coefficients save 550·4=2200 bytes of storage.

4.1.4 Suggested improvements

Since the implicit partial enumeration was unable to produce a solution within the real-time requirements, see Table 4.1, other improvements were considered. Bazaara and Elshafei in [4.6.2] investigated several methods to accelerate the tree search by introducing additional fathoming criteria. Strong fathoming is defined to be those assignments that are prohibited due to the constraints. Weak fathoming is for those assignments in which the maximum possible objective function is less than the current value. These acceleration methods involve weak fathoming to produce a sub-optimal solution.

One method involves expressing the objective function in terms of the distance between two items and cost of locating an item, the applying a "ranking" algorithm. This technique does not consider constraints and could not be applied here. A "steepest descent" on the tree could be considered by ranking the coefficients, but would require the recomputing of the "descent" matrix from the original cost matrix of size n(n-1)/2 values after every assignment and searching the matrix with mk values, where m is the

number of unassigned items, to produce the optimal assignment. Io may be required to save the matrix status to facilitate backtracking.

Another method is to define an "alpha" factor, α, such that weak fathoming will occur whenever the predicted objective funtion is less than α times the current maximum, i.e., if z_o is less than alpha time the last value of z_o where $\alpha>1$, then fathom. An α that is too large would result in generating only a few nodes. Relaxation of the α-factor, when a satisfactory assignment could not be found, would regenerate too many of the same nodes. Other sub-optimal solution such as cluster analysis under constraints could produce solutions with less overhead and is considered in the next section.

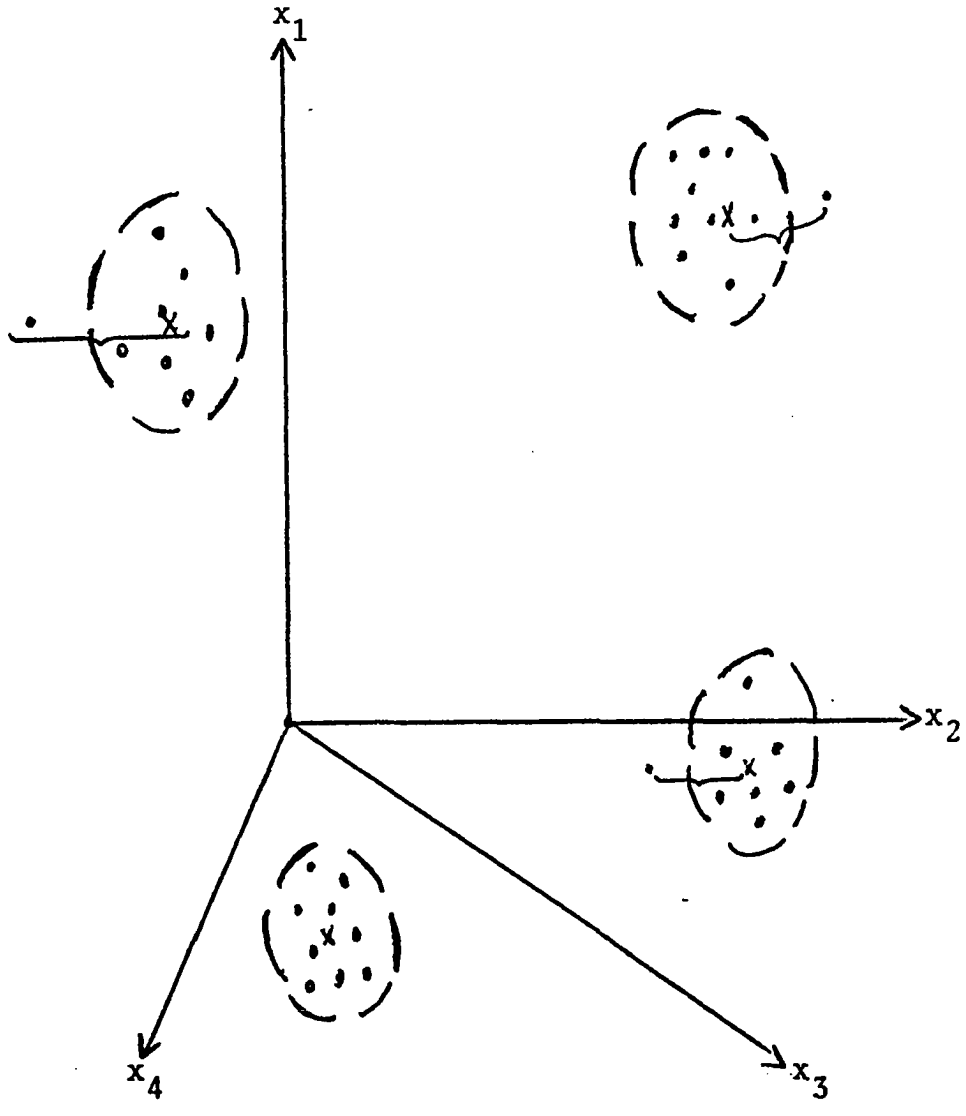
4.2 Cluster Analysis

Cluster analysis is used to classify objects into categories where little is known about the structure of the category, see Figure 4.8. From sorting the objects into groups so that the degree of association among the members of the same group will be as high as possible, the user extracts information concerning the group structure or interrelationships of the members of the group. The ability to partition the elements into distinct categories depends heavily on the method used to measure the association of an element to a group and the technique used to give an a

Figure 4.8 Clusters representation with respect to an n-space using the "nearest" centroid method of assigning cluster members.

· programs

x cluster centroid



priori description of the group.

Cluster analysis was considered as a heuristic method for assigning n programs to k processing elements so as to maximize the amount of communication between the programs within a given processing element, which in the problem considered in this thesis is equivalent to minimizing the bus traffic due to inter-program communication. There are basically two approaches to cluster analysis, hierarchical and non-hierarchical.

Hierarchical cluster analysis involves the use of a correlation matrix to perform the assignments especially when the number of clusters have not been predetermined and is better suited to generation of classification schemes. Non-hierarchical cluster methods depend upon the number of clusters being known; they are used mainly for assignment problems. The basis of the non-hierarchical cluster algorithms consists of choosing data units to describe the set of objects, initializing the centroid of each cluster, selecting a distance function to determine the "nearness" of a point to a centroid, and the method for updating the cluster centroid, see Figure 4.8.

Since little research into the use of the cluster analysis has been applied to situations where there are physical constraints on the clusters (i.e., density, size, number of elements), the basic variations in the algorithms were compared to determine the best algorithm that could be

used to compute a sub-optimal solution to assign programs to processing elements in order to maximize the intra-processing element communication using different job mixes.

4.2.1 Selection of the data units and the distance function

In many situations the observations can be described as points in n-space, where each coordinate component represents a measurable feature of the observation. The Euclidean distance formula can be used to assign the point to that cluster whose distance from the centroid is a minimum. In cases where the coordinates are measured in different units; e.g., inches, pounds, seconds, the units may have to be weighted by "cost" coefficients in order to provide better separation of the cluster centroids. In problems where the space is not a metric space (e.g., those involving pattern recognition) the point is closest to that centroid which has the greatest agreement among the coordinates. In such problems, the distance measure does not satisfy the triangle inequality. Several nonmetric distance functions have been investigated. Two such functions were evaluated by Lance and Williams. These are related to the entropy function which will produce a minimal value for those points having the greatest amount of agreement among the components, i.e., the distance will be small for those points whose coordinate differences are small. For binary data, the "informatic" function measures the degree of similarity:

$$d(a,b) = \Sigma \frac{|a_i - b_i|}{(a_i + b_i + 1)} \qquad \text{(eq. 4.3)}$$

For example, let a=(1,1,0,0), b=(1,0,1,0), and c=(0,0,1,1), then d(a,b)=1 and d(a,c)=2 which shows that there is greater agreement between items a and b than between a and c.

A similar function to that of eq. 4.3 was developed for non-negative data:

$$D(a,b) = \frac{\Sigma . |a_i - b_i|}{\Sigma\ (a_i + b_i)} \qquad \text{(eq. 4.4)}$$

For example, let a=(1,2,1,0), b=(0,2,1,1), and c=(1,0,0,2), then D(a,b)=1/4 and D(a,c)=5/7. Therefore, a and b are more in agreement than a and c. The distance between b and c is the same as the distance between a and c, i.e., D(b,c)=5/7. Another function that is sometimes used is:

$$e(a,b) = \Sigma\ |a_i - b_i| \qquad \text{(eq. 4.5)}$$

where the values of e for the previous example are e(a,b)=2, e(a,c)=5, and e(b,c)=5.

The experiments of Lance and Williams showed that the informatic function was better than the Euclidean distance formula and the distance function in eq. 4.5 for creating well-defined clusters with good separation between centroids. Care must be taken in the use of eq. 4.4 which may produce a negative distance for the cluster algorithms which use translation or rotation about the centroids.

For the problems considered in this thesis, the description of a program is given by a vector each component of which represents the amount of time required for a message to be processed, if it were sent over the bus to another processing element. Each component represents a different message as shown in Table 1.1. Each program will have resource requirements, such as the amount of run-time required and the number of pages of memory needed as shown in Table 1.2. Equation 4.4 was used as the assignment criteria with the objective to maximize the amount of intra-processor communication. The constraints on the cluster will be the total amount of memory of the processor and the length of a standard time frame. The next subsections will describe the variations in the cluster algorithm and compare the results of the cluster analysis with constraints added and for different job mixes.

4.2.2 Centroid initialization

The second step in arriving at the cluster algorithm is to determine the method to initialize the cluster centroids, called seeds. A set of k seed points can be used as the cluster centroids around which the set of n data points can be grouped. The following methods have been suggested as techniques for generating the initial centroids [4.6.19]:

1. Choose the first k data points in the set. If the initial configuration does not "bias" the centroid in a way that affects the

convergence of the centroid, i.e., does not allow the centroid to move away from the initial centroid as more data points are added to the cluster, then this is the most efficient method. If the data is not ranked by any method, then this is essentially a random initialization, but has the advantage that the centroid will be a member of the data set.

2. Partition (according to the criteria of the user) the data into k mutually exclusive groups and compute the initial seed for the cluster centroids.

3. In 2, choose as the seed points the ones that form a basis for the vectors and span the vectors in a manner similar to that of a vector space. Also, the initial seed point must be well separated from each other, but not isolated points. For example, order the data points by density and choose the one with the highest density as the initial seed point. Choose the remaining seed point in descending order of the density. (Density can be defined by the user, e.g., "the most representative" criteris.)

4. Subjectively select data points as seed points. For example, rank the data on the component of the feature vector that is the most heavily utilized and initialize the centroid to zero but set that component to the feature weight found.

The above methods 1, 3, and 4 are included in the CLUSTER procedure in Appendix A; where method 1 uses the first k programs, 3 uses the programs with the most IO, and 4 uses the heaviest utilized message. There are other methods for initializing centroids and using reflection of the centroids to generate subsequent centroids that are described by Anderberg [4.6.19]. In the present context, the points in the space to be clustered represent the computer programs under consideration.

4.2.3 Nearest centroid sorting

Nearest centroid sorting is one method for determining cluster membership and is used in both hierarchical and non-hierarchical cluster analysis. After starting with an initial set of seed points as cluster centroids, the clusters are constructed by assigning elements to the cluster with the nearest seed point, subsequently recomputing the centroids until a stable configuration occurs, and reassigning the points. The stability criteria can be: no elements were reassigned after an iteration, the centroids were unaffected during the iteration, or the total distances from the respective centroids is less than some threshold value.

There are two methods of iterating and updating the centroids during cluster analysis. These are Forgy's and Mac Queen's algorithm.

FORGY'S ALGORITHM:

1. Begin with any desired initial set of centroids.
2. Allocate each data element to the cluster with the nearest centroid where the centroid remains constant until all points have been assigned.
3. If no element has changed their cluster assignment or the maximum number of iterations has been reached, then terminate the algorithm.
4. Compute the new centroid, i.e., the components of the centroid will be the average of the components of the elements assigned to that cluster. Go back to step 2

Experience with this algorithm has determined that for the cases investigated convergence will occur after five iterations or less. The maximum number of iterations required seldom went longer than ten. For clustering under constraints, the maximum number of iterations was set to ten for the FORGY method included in the CLUSTER procedure. In most instances, the number of iterations required was from four to six iterations with the greatest change in the total distance from the centroids occuring in the second and third repetitions and only scant change after six.

For n points (i.e., programs) and k clusters (i.e., processing elements), each assignment iteration of the algorithm requires nk distance computations and n(k-1) comparisons of distances. Since k is usually much smaller than n and the number of iterations needed is small, it is cheaper to investigate several initial values for centroids than to use hierarchical cluster analysis, see [4.6.19].

Mac Queen's k-means method recomputes the cluster centroid according to the current cluster membership rather than at the end of an iteration.

MAC QUEEN'S ALGORITHM:

1. Take the first k elements as seed points or use one of the other initialization methods.
2. Assign the points to the cluster with the nearest centroid. After each assignment, recompute the centroid.

3. After step 2, make one more pass through the set of elements assigning the elements to the nearest centroid and update the centroid after each assignment. (Updating the centroid at this step is a variant of the original Mac Queen method.)

The convergence of this algorithm depends heavily on the centroid initialization criteria and on the order the elements are processed. If the cluster centroids are well separated, then the order has little effect except in cases where the element falls between two clusters.

With k clusters and n elements, the number of distance calculations required would be nk and the number of distance comparisons would be n(k-1) with n centroid updates for each iteration. Fewer iterations are used than in the Forgy method. If the user starts with k points as the initial seed points, then only nk+k(n-k) distance calculations, (k-1)(2n-k) distance comparisons, and n-k+n centroid updates are required. For n=50 and k=5 processors and using the Mac Queen method, the number of distance calculations are 475, the number of comparisons are 380, and the centroid updates are 95 which is better than the Forgy method requirements of 1250, 500, and 2500 for the average of 5 iterations. Therefore, the Mac Queen method is cheaper than the Forgy method as well as the hierarchical cluster methods. For the purpose of investigating the rate of convergence of the MACQUEN method in the CLUSTER procedure in Appendix A,

the maximum number of iterations allowed was set at four and updating the centroid continued at each assignment within each iteration. Even clustering under the constraints (i.e., the inequality constraints of eq. 3.1) the centroids converged after two iterations with very few requiring three.

4.2.4 Clustering under constraints

Due to the constraints, the clusters can become full, e.g., a program is unable to fit within the remaining storage locations, and it is impossible to assign the remaining elements. The CLUSTER procedure has the ability to increase the number of clusters. This has the added value of being able to reconfigure the system when a processor is added or fails. One method for choosing the new centroid is to initialize the centroid to the same value as the centroid of any one of the clusters. Elements that are assigned to the new cluster will be the ones that are closer to the centroid and/or that will not fit in any other cluster. The latter case will cause the most noticeable change in the centroid and its distance from the initial centroid will increase. This will cause the centroid to converge away from the initial centroid. This is an inexpensive method to increase the number of clusters without having to start the reclustering at the point of reinitializing the seed points. After the number of clusters is determined, the user can recompute the cluster centroids

using the last computed centroids as the initial seed points, and the method is still less costly than using hierarchical cluster analysis to determine the number of clusters.

4.2.5 Unknown initial number of clusters

One method to perform non-hierarchical cluster analysis when the number of clusters are not known is a heuristic variation of Ball and Hall's "lumping" technique in the ISODATA system [4.6.19]. Basically the algorithm starts off with every point as the centroid, merges the clusters until nc more merges can take place, and recomputes the centroid after every merge.

HEURISTIC ALGORITHM:

1. Start off with each data point as a cluster centroid.
2. Compare cluster centroids and merge the two clusters together whose distance between their centroids is the closest. Merging consists of removing all of the elements of one cluster and reassigning them to the other, then freeing the empty cluster. The centroid is updated to reflect the utilization of the component by the members of the cluster, i.e., the new component of the centroid will be the maximum of the old centroid and the elements of that clusters as in eq. 4.5.
3. The algorithm terminates when it is impossible to merge two clusters.

A variant would be to use averages as in the other methods if the centroids tend to become identical. This algorithm appears to have an advantage over the other methods provided that the number of elements to be clustered is small, say $k \leq 1/4n$.

For the heuristic algorithm, the approximate number of distance calculations is $E_i(n) \cdot (E_i(n)-1)$ and the number of distance comparisons is approximately $E_i(n) \cdot (E_i(n)-2)$ with $E_i(n)$ as the expected number of clusters during the i^{th} iteration, where $E_i(n) = \sum_j p_{ij} n_j$ and p_{ij} is the probability that the cluster n_j is not merged during the i^{th} iteration. The n is the initial number of clusters.

The heuristic algorithm performs best under the tightly constrained configurations with only large programs. It is possible that this algorithm, called HEUR in Appendix A, will generate more clusters than actually needed. Combining this algorithm, to obtain the initial number of clusters, and one of the previous cluster methods will still be more efficient than hierarchical cluster analysis with the savings in recomputing the correlation matrix.

4.3 Comparison of the Cluster Algorithms

Each algorithm, FORGY and MACQUEN in the CLUSTER procedure was executed using three different initialization schemes and three different job mixes. Criterion for evaluating the performance of the algorithms were: minimal

execution time of the algorithm and minimization of the amount of inter-processing element communication (or minimize the total distance). The storage requirement for each algorithm is essentially the same. The distance function selected was eq. 4.4 based on the findings of Lance and Williams [4.6.20].

The initialization of the centroids using method 1 and method 3 will be referred to as FIRSTN and BUSYPR, respectively, in Tables 4.2 and 4.3 as well as in the CLUSTER procedure. Method 4 will use the most heavily weighted and utilized message as the centroid ranking criteria and is called BUSYMSG. BUSYPR and BUSYMSG requires more execution time than FIRSTN in order to rank the programs.

Job mixes were generated to represent three situations:

1. LARGE programs, a few large programs with heavy inter-communication and tightly constrained within the processing elements.
2. SMALL programs, many small programs with only a few interprogram messages to be handled by each.
3. MIXED, a combination of 1 and 2 with approximately ten percent of the programs that are large.

In order to evaluate the cluster algorithms, the above job mixes required sixty percent of the total bus capacity and seventy percent of the processor run-time and of the memory. When the programs use most of the systems resources, the clusters are too biased by the initialization causing little

TABLE 4.2 Gain in Intra-processing Element Communications

iteration method	initialization technique	job mix types LARGE*	MIXED+	SMALL+
FORGY	FIRSTN	26	620	391
	BUSYPR	29	613	449
	BUSYMSG	33	617	354
MACQUEN	FIRSTN	33	619	395
	BUSYMSG	29	588	395
	BUSYMSG	42	620	407

		order n*k
*	LARGE	50
+	other	250

TABLE 4.3 Clustering Algorithm Run-time Estimates
(average times in ms)

program sample	order		firstn		busypr		busymsg	
			forgy	macquen	forgy	macquen	forgy	macquen
LARGE	50	initialization time	10	10	10	20	10	10
		iteration time	220	200	240	210	280	250
		number of iterations	2	2	2	2	3	3
MIXED	250	initialization time	10	10	30	20	20	20
		iteration time	880	820	730	690	860	790
		number of iterations	4	3	3	2	4	3
SMALL	250	initialization time	10	10	20	30	10	20
		iteration time	980	970	1200	940	840	630
		number of iterations	5	3	6	4	3	2
		TOTAL average run-time	703	673	743	636	676	574

average initialization time

FIRSTN	10
BUSYPR	12.5
BUSYMSG	15

average method time

FORGY	1038.33
MACQUEN	916.67

change in the composition of the cluster from iteration to iteration. Other problems of a tightly constrained system result in the failure of the algorithm to assign all of the programs or result in the convergence of the algorithm too quickly to change the initial centroids.

Table 4.2 shows the gain (or savings) in inter-processing element communication resulting from the assignment of the programs to the processors using each cluster method. The two methods that produce good results are the FORGY algorithm with BUSYPR centroid initialization and the MACQUEN algorithm with BUSYMSG initialization. The difference in the average gain in intra-processor communication is only ten time units. The MACQUEN-BUSYMSG method does perform better in the cases that have large programs with heavy interaction of the message traffic which might be expected considering the initialization technique. In Table 4.3 this method had the smallest average run-time requirements for all the job mixes which again might have been anticipated since the MACQUEN method requires fewer iterations (especially for cases involving many small programs).

4.4 Cluster Analysis and Quadratic Programming

For n=6 programs with 4 messages and 3 processors of the example in Chapter 1, the quadratic zero-one programming algorithm found the optimal solution which had a maximum objective function value of 20 time units in 27 sec. on the

IBM 370/158. This example was so tightly constrained that many of the cluster analysis methods were unable to complete. Only 36 nodes out of the 218 were possible solutions. The initialization method biased the outcome so much that the assignment generated in the first iteration was the final solution. All six cluster methods executed in 27 sec., but only the two methods using FIRSTN initialization generated a solution of 10 time units for the intra-processor communication.

For the LARGE job mix problem, n=10 programs with 10 messages and k=5 for order of 50, the value of the objective function was 39 after 1 min. of execution time for the QUAD procedure. Only an improvement of 1 time unit was reached by increasing the run-time to 10 min. The only cluster method that was able to improve on this was the MACQUEN-BUSYMSG which had a savings of 42 time units in the intra-processor communication. The range of the other methods are from 26 to 33 time units, see Table 4.1. The system could afford to execute all six methods for the three different job mixes and choose the best result, since the execution time was less than 30 sec.

For the MIXED and the SMALL sample job mix, both of order 250, the QUAD procedure generated a sub-optimal solution after 3 min. with the value of the objective function equal to 300 and 198 time units, respectively. All of the cluster methods were able to produce better results.

Therefore, the best applications of the quadratic programming algorithm are those involving the utilization of most of the system resources. In the cases that many of the processors have failed and cannot be clustered, it would be better to have even a sub-optimal solution from the quadratic zero-one programming algorithm.

One attempt to improve the performance of the quadratic programming algorithm was to use the cluster routine as a method to initialize the root node of the implicit partial enumeration algorithm. The combination produced a result that was somewhat better than the quadratic binary programming could produce alone, see Table 4.4. The improvement was not significantly greater than the result obtained by cluster analysis alone. This is possibly due to the fact that the initial node is beyond the tree node for an optimal solution, i.e., the appropriate branch will not be generated during a backtrack and the tree is too large to generate the permutation.

TABLE 4.4 Results of Combining Cluster Analysis and QUAD
(with loose constraints, i.e., 75% resource utilization)

Interaction Type*	Order	Explicit Constraints	Memory Requirement	IBM 370/158 Run-time	No. of Sub-optimal Sol'n Generated	Value of the Obj. Funct. Cluster/QUAD
25% heavy 75% light	250	12	144k	16.36 sec	2	619/620**
100% heavy	50	12	72k	16.22 sec	4	42/47**
100% light	250	12	144k	16.18 sec	1	392/402**

** Did not improve after 20 min. exec. time.

* In terms of the amount of communication between programs.

4.5 References

1. Bazaara, M. S. and Goode, J. J., "A cutting plane algorithm for the quadratic set covering problem." (To appear in ORSA.) ORSA vol. 23, no. 1, January-February 1975.
2. Bazaara, M. S. and Elshafei, A. N., "Exact and Heuristic procedures for the quadratic assignment problem." (Working paper).
3. Carson, R. C. and Nemhouser, G. L., "Scheduling to Minimize Interaction Costs." Oper. Res. vol. 14, no. 1, 1966, pp. 52-58.
4. Garfinkel, R. S. and Nemhouser, G. L., Integer Programming. John Wiley, 1972.
5. Gilmore, P. C., "Optimal and Suboptimal Algorithms for the Quadratic Assignment Problem." J. Soc. Indust. Appl. Math., vol. 10, 1962, pp. 305-313.
6. Goeffrion, A. M., "An improved implicit enumeration approach for integer programming." The RAND Corporation Memorandum, RM-5644 PR, June 1968.
7. Goeffrion, A. M., "An improved implicit enumeration approach for integer programming." Oper. Res. vol. 17, 1969, pp. 437-454.
8. Glover, F., University of Colorado, Boulder, Colorado. Personal communication, March 1975.

9. Hammer, P. L. and Hansen, Pierre, "Quadratic zero-one programming." (Working paper, April 1972).

10. Hanan, M. and Kurtzberg, J. M., "A review of quadratic assignment problems." SIAM Review vol. 14, no. 2, April 1972, pp. 324-342.

11. Ibaraki, T., Liu, J. K., Baugh, C. R., and Muroga, S., "An Implicit Enumeration Program for Zero-One Integer Programming." International Journal of Computer and Information Sciences, vol. 1, no. 1, 1972, pp. 75-92.

12. Laughhunn, D. J., "Quadratic binary programming with applications to Capital Budgeting Problems." pp. 454-461.

13. Lawler, E. L., "The Quadratic Assignment Problem." Mgt. Sci. vol. 9, July 1966, pp. 586-599.

14. Lutz, Devine, Kumin, and Smith, "An application of operations research to school desegregation." Mgt. Sci., December 1972.

15. Pierce, J. F. and Crowstron, W. B., "Tree Search Algorithms for Quadratic Assignment Problems." Naval Research Logistics Quarterly, vol. 18, 1971, pp. 1-36.

16. Ross, G. T. and Soland, R. M., "Branch and Bound Algorithm for the Generalized Assignment Problem." Mathematical Programming, (8) 1975, pp. 91-103.

17. Taha, H. A., Oper. Res., an introduction. Macmillian, 1972.

18. White, W. W., "Computing Algorithms for Mathematical Programming." Comp. Surveys vol. 15, no. 3, September 1973.

cluster analysis:

19. Anderberg, Michael R., Cluster Analysis for Applications. Academic Press, 1973.

20. Lance, G. N. and Williams, W. T., "Computer programs for Hierarchical Polythetic Classifications. (Similarity Analysis)." Computing Journal, vol. 9, no. 1, 1966.

Karnaugh map:

21. Kohavi, Z., Switching and Finite Automata Theory. McGraw-Hill, 1970.

CHAPTER 5

SIMULATION OF THE TWO-LEVEL HIERARCHICAL CONTROLLER

The purpose of the simulation of the hierarchical controller for the distributed network of computers is to determine the performance levels as a function of various tasks that the operating system will have to execute. For example; the "cost" of reconfiguring the system in terms of the distributed system being unable to complete the processes assigned to it. The maximum frequency at which reconfiguration can take place is at the rate of every standard time frame, but a new strategy may be needed when the overhead is too great or the improvement in the performance index is too small.

The simulation for this thesis was written in PL/1 using the CRETIN simulation package [13]. This package was especially prepared to use the dynamic storage allocation, the list processing techniques, and the data structures available in PL/1.

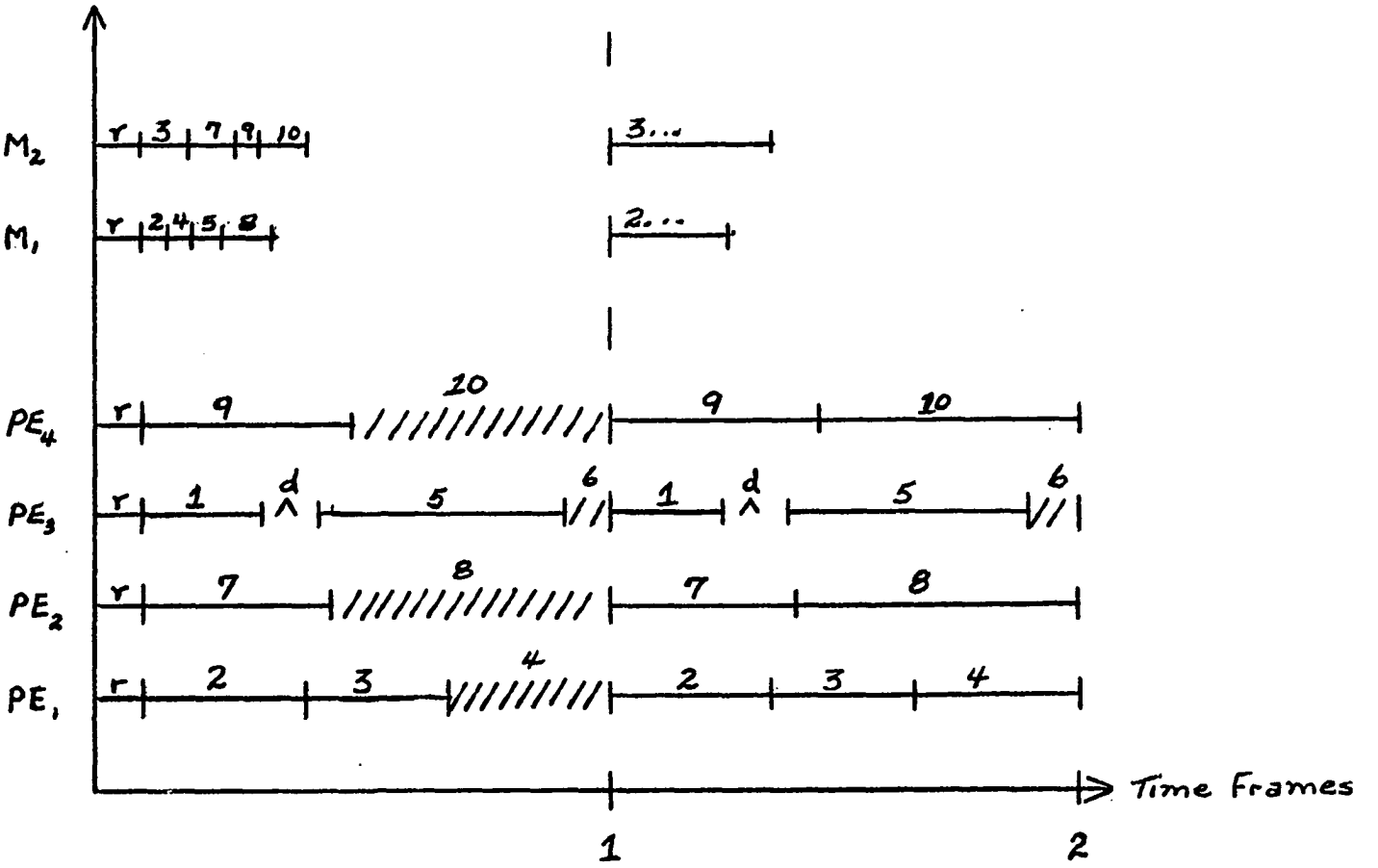
The simulation model (the data cards for the model are described in Appendix B) is a "worst case" situation for the

tightly constrained system of order 40 that was used to compare the QUAD procedure with cluster analysis. Ten percent of the standard time frame is charged as the "overhead" due to reconfiguration and scheduling, i.e., the time frame is 100 time units and the overhead is ten time units.

In Figure 5.1a, the initial reconfiguration and scheduling overhead (r) takes ten time units. Since all of the processing elements are scheduled by the MACQUEN-BUSYMSG method, the entire standard time frame is needed just to compute the program assignment. For processing elements 1, 2, and 4, the programs that were unable to be completed (due to the ten unit time loss) are 4, 8, and 10, respectively. Since the schedule is not optimal, program 1 was delayed ten units until message 10 was transmitted. The savings in bus traffic as a result of the scheduling was 12 time units (messages 1 and 6 were not transmitted over the bus).

Figure 5.1b shows the case where processing element 1 failed during the second standard time frame. The failure occurred after program 2 had executed for ten time units. During the reconfiguration, cluster analysis was unable to assign all of the programs and generated an interrupt. Processing elements continued, but without a complete schedule. Processing element 1 recovered 3 frames later, i.e., between frame 5 and 6, but it is unable to resume processing until reconfiguration takes place at the start of frame 7.

Figure 5.1a Gantt Chart with Reconfiguration Overhead



//// - unable to complete
d - waiting for message
r - reconfiguration and scheduling overhead

Figure 5.1b Gantt Chart for a Processing Element Failure

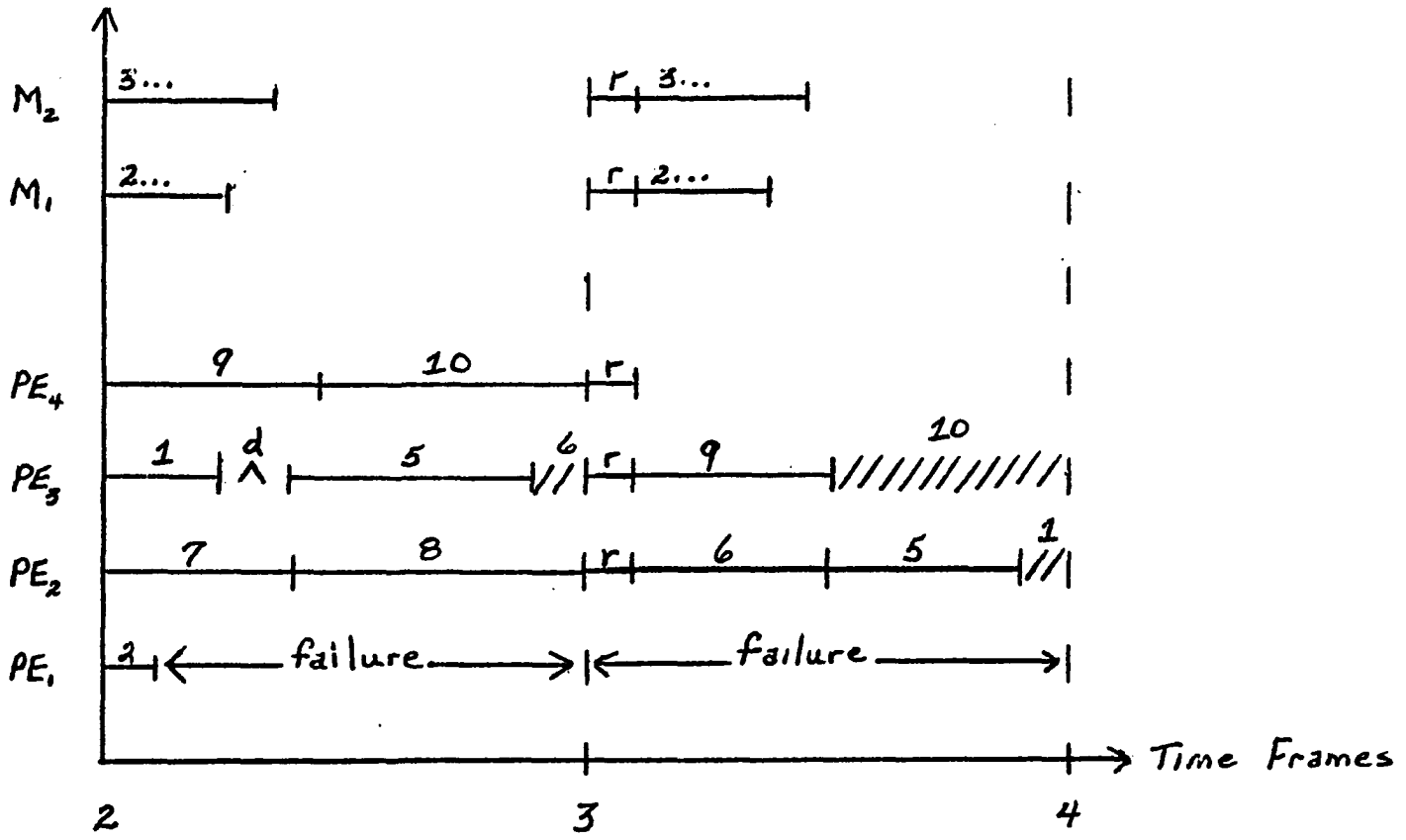


Figure 5.2 Processing Element Schedule

PE_i	Schedule	Run-time Required
1	2,3,4	100
2	7,8	100
3	1,5,6	100
4	9,10	100

This miniature model of a tightly constrained situation serves to illustrate that reconfiguration can reduce the amount of missed deadlines. Further missed deadlines could occur if messages 1 and 6 were included in the bus traffic (causing a delay of at least ten additional time units and at most a delay of 17 time units).

The main goal of the simulation was to show that reconfiguration "policies" could be generated within the real-time constraints placed on the system environment. The amount of "work" accomplished by the processing element is definitely affected by the reconfiguration algorithm, the scheduling algorithm, and the overhead. Different arrangements of these factors can greatly change the performance.

The main purpose of this dissertation was to present the applicability of control theory concepts to an operating system design (and not the simulation) for a particular system, the distributed network of computers. With more accurate system specification for the hardware and the software overhead, this model can be extended in a variety of ways. Specifically, a more detailed scheduling algorithm is needed to be expressed in terms of the message traffic as well as the deadlines. (This was observed in the simulation that showed unnecessary delays for message waiting.) Also, different heuristics can be built-in so that reconfiguration can take place in idle processing elements and reduce overhead.

BIBLIOGRAPHY

1. Agerwala, T. and Flynn, M., "Comments on capabilities, limitations and 'Correctness' of Petri nets." Proc. on conf. on Computer Architecture, 1973, pp. 81-86.

2. Anderberg, M. R., Cluster Analysis for Applications. Academic Press, 1973.

3. Anderson, G. A., "Interconnecting a distributed processor for Avionics." First Annual Symposium on Computer Architecture, 1973.

4. Bazaara, M. S. and Goode, J. J., "A Cutting Plane Algorithm for the Quadratic Set Covering Problem." ORSA, vol. 23, no. 1, January-February 1975.

5. Bazaara, M. S. and Elshafei, A. N., "Exact and Heuristic Procedures for the Quadratic Assignment Problem," (Working paper, 1975).

6. Baer, J. L., "A survey of some theoretical aspects of multiprocessing." ACM Computing Surveys, vol. 5, no. 1, March 1973, pp. 30-80.

7. Baer, J. L., "Models for the design, simulation, and performance of distributed-function architecture." Computer, vol. 7, no. 3, March 1974, pp. 25-30.

8. Bell, C. G. and Newell, A., Computer Structures; Readings and Examples. McGraw-Hill, 1971.

9. Carson, R. C. and Nemhouser, G. L., "Scheduling to Minimize Interaction Costs." Oper. Res. vol. 14, no. 1, 1966.

10. Chen, R. C., Bus Communication System. PhD dissertation, Carnegie-Mellon, January 1974.

11. Coffman, E. G. and Denning, P. J., Operating Systems Theory. Prentice-Hall, 1973.

12. Dijkstra, E. W., "The structure of 'THE' multiprogramming System." CACM, vol. 11, no. 5, May 1968, pp. 341-345.

13. Edwards, J. and Fox, S., CRETIN simulation package. University of Oklahoma, TR-1974-1.

14. Farber, D. J., Feldman, J., Heinrich, F. R., Hopwood, M. D., Larson, K. C., Loomis, D. C., and Rowe, L. A., "The distributed computing system." Proc. COMPCON, February 1973, pp. 31-34.

15. Flynn, M. J., "Some computer organizations and their effectiveness." IEEE trans. on Comp., vol. c-21, no. 9, September 1972, pp. 948-960.

16. Garfinkel, R. S. and Nemhouser, G. L., Integer Programming. John Wiley, 1972.

17. Gilmore, P. C., "Optimal and Suboptimal Algorithms for the Quadratic Assignment Problem." J. Soc. Indust. Appl. Math., vol. 10 (1972), pp. 305-313.

18. Goeffrion, A. M., "An improved implicit enumeration approach for integer programming." The RAND corporation Memorandum, RM-5644 PR, June 1968.

19. Goeffrion, A. M., "An improved implicit enumeration approach for integer programming." Oper. Res. vol. 17, 1969, pp. 437-454.

20. Glover, F., University of Colorado, Boulder, Colorado. Personal communication, March 1975.

21. Goodell, W., "The Minis Gang up." Computer Decisions, vol. 6, no. 6, June 1974.

22. Hammer, P. L. and Hansen, Pierre, "Quadratic zero-one programming," (Working paper, April 1972).

23. Hanan, M. and Kurtzberg, J. M., "A review of Quadratic Assignment Problems." SIAM Review, vol. 14, no 2, April 1972, pp. 324-342.

24. Hansen, P., "Programming methodology for operating systems design." IFIP (proc.), vol. 58, no. 1, January 1970, p. 111.

25. Hellerman, L., "A measure of Computational Work." IEEE trans. on Comp., vol, c-21, no. 5, May 1972.

26. Horning, J. J. and Randell, B., "Process Structuring." ACM Computing Surveys, vol. 5, no. 1, March 1973.

27. Hvarinen, L., Mathematical Modeling for Industrial Processes. Springer-Verlag Notes on Information and Economic Models, no. 19, 1970.

28. Ibraki, T., Liu, J. K., Baugh, C. R., and Muroga, S., "An Implicit Enumeration Program for Zero-one Integer Programming." Int. J. of Comp. and Info. Sci., vol. 1, no. 1, 1972, pp. 75-92.

29. Jansen, E. E., "A distributed function computer for real-time control." Second Annual Symposium on Computer Architecture, 1973.

30. Johnson, D., "A process-oriented model of resource demands in large multiprocessing computer utilities." PhD dissertation, University of Texas at Austin, 1972, pp. 26-44.

31. Jordan, J. W., "Task scheduling for a real-time multiprocessor." NASA TN B-5786, May 1970.

32. Kirk, D. E., Optimal Control Theory, an Introduction. Prentice-Hall, 1970.

33. Kohavi, Z., Switching and Finite Automata Theory. McGraw-Hill, 1970.

34. Lance, G. N. and Williams, W. T., "Computer programs for Hierarchical Polythetic Classifications. (similarity analysis)" Computing Journal, vol. 9, no. 1, 1966.

35. Laughhunn, D. J., "Quadratic binary programming with applications to Capital Budgeting Problems." Oper. Res., vol. 18, no. 3, March 1970, pp. 454-461.

36. Lawler, E. L., "The Quadratic Assignment Problem." Mgt. Sci. vol. 9, July 1966, pp. 586-599.

37. Lee, T. H., Adams, G. E., and Gaines, W. M., Computer Process Control: Modeling and Optimization. John Wiley, 1968.

38. Lewis, J., editorial, IEEE trans. on Automatic Control, Newsletter, December 1970.

39. Lowe, T. C., "Analysis of an Information system Model with Transfer penalties." IEEE trans. Comp., vol. c-22, no. 5, May 1973, pp. 469-472.

40. Lutz, Devine, Kumin, and Smith, "An application of Operations Research to School Desegregation." Mgt. Sci., vol. 19, no. 4, December 1972.

41. Mesarovic, M. D., "Multilevel systems and concepts in Process Control." IEEE (proc.), vol. 58, no. 1, January 1970, p. 111.

42. McGandy, T. P., Stochastic Control and State Estimation. John Wiley, 1974.

43. Miller, R. E., "A Comparison of some theoretical models of Parallel Computation." IEEE trans. on Comp., vol. c-22, no. 8, August 1973, pp. 718-727.

44. Nisnevich, L. and Strasbourger, E., "Decentralized Priority Control in Data Communications." Second Annual Symposium on Computer Architecture, January 1975, pp. 1-6.

45. Noe, J. D. and Nutt, G. J., "Macro E nets for representation of Parallel Systems." IEEE trans. on Comp., vol. c-22, no. 8, August 1973.

46. Northouse, R. A., "Dynamic scheduling of large digital computer systems using adaptive control and clustering techniques." PhD dissertation, Purdue University, June 1972.

47. Northouse, R. A., and Fu, K., "Dynamic scheduling of large digital computer systems using adaptive control and clustering techniques." IEEE trans. on Systems, Man, and Cybernetics, vol. smc-3, no. 3, May 1973.

48. Pierce, J. F. and Crowstron, W. B., "Tree Search Algorithms for Quadratic Assignment Problems." Naval Research Logistics Quarterly, vol. 18, 1971, pp. 1-36.

49. Ramamoorthy, C. V., Chandy, K. M., and Gonzalez, Jr., M. J., "Optimal scheduling strategies in a multiprocessor system." IEEE trans. on Comp., vol. c-22, no. 2, February 1972, pp. 137-146.

50. Ross, G. T. and Soland, R. M., "Branch and Bound Algorithm for the Generalized Assignment Problem." Math. Prog. vol. 8, 1975, pp. 91-103.

51. Salton, G., Automatic Information Organization and Retrieval. McGraw-Hill, 1968.

52. Savas, E. S., Computer Control of Industrial Processes. McGraw-Hill, 1965.

53. Schoeffler, J. D., "On-line multilevel systems." Wismer, D. A. ed., Optimization methods for large-scale systems. McGraw-Hill, 1971.

54. Sorenson, P. G., "Interprocessor Communication in Real-time Systems." IEEE Symposium on Computer Software Reliability, January, pp. 1-7.

55. Taha, H. A., Oper. Res., an Introduction. Macmillian, 1972.

56. Thurber, K. J., "A systematic approach to the design of digital busing structures." Proc. AFIPS conf., vol. 41, (FJCC'72) pp. 719-740.

57. Thurber, K. J. and Jack, L. A., "Time Driven Scheduling." (proc.) COMPCON 1974, pp. 181-185.

58. White, W. W., "Computing Algorithms for Mathematical Programming." ACM Computing Surveys, vol. 15, no. 3, September 1973.

APPENDIX A

Quadratic zero-one programming
procedure QUAD

Cluster analysis data structures
and procedure CLUSTER

APPENDIX A

A.1 Contents of the Cluster Analysis Data Structures

The cluster structure, CLUSTER, contains a pointer to the next cluster, CLPTR, and a pointer to the first program on the cluster list, PRPTR. It also contains three integer quantities: a count of the number of programs in the cluster, CLKNT; the cluster number, NUM; and the dimension of a dynamically allocated array, NC. The array is of length equal to twice the number of messages plus four. (This will contain the centroid, the current sum of the messages of the programs assigned to that cluster, the bounds on the cluster, and the amount of resources allocated to the programs in the cluster.)

The program descriptor element contains two pointers: one to indicate the next program in the list to be assigned, FRPTR, and the other to indicate the next program in the cluster, CLUST. It has three integers to indicate the program number, NUMB; the current cluster assignment number, CLNO; and the dimension of the dynamic storage array, NP for the number of messages plus two bounds. The array contains the amount of message time needed for each message by that program and the amount of run time as well as the amount of memory required.

These structures are used in the FORGY and MAC QUEEN cluster methods as well as the heuristic method; these algorithms are given in Chapter 4. (The structures for the QUAD program are also discussed in Chapter 4.) The following Figures A.1 and A.2 show the linkages of programs onto the clusters.

Figure A.1 Composite Data Structure

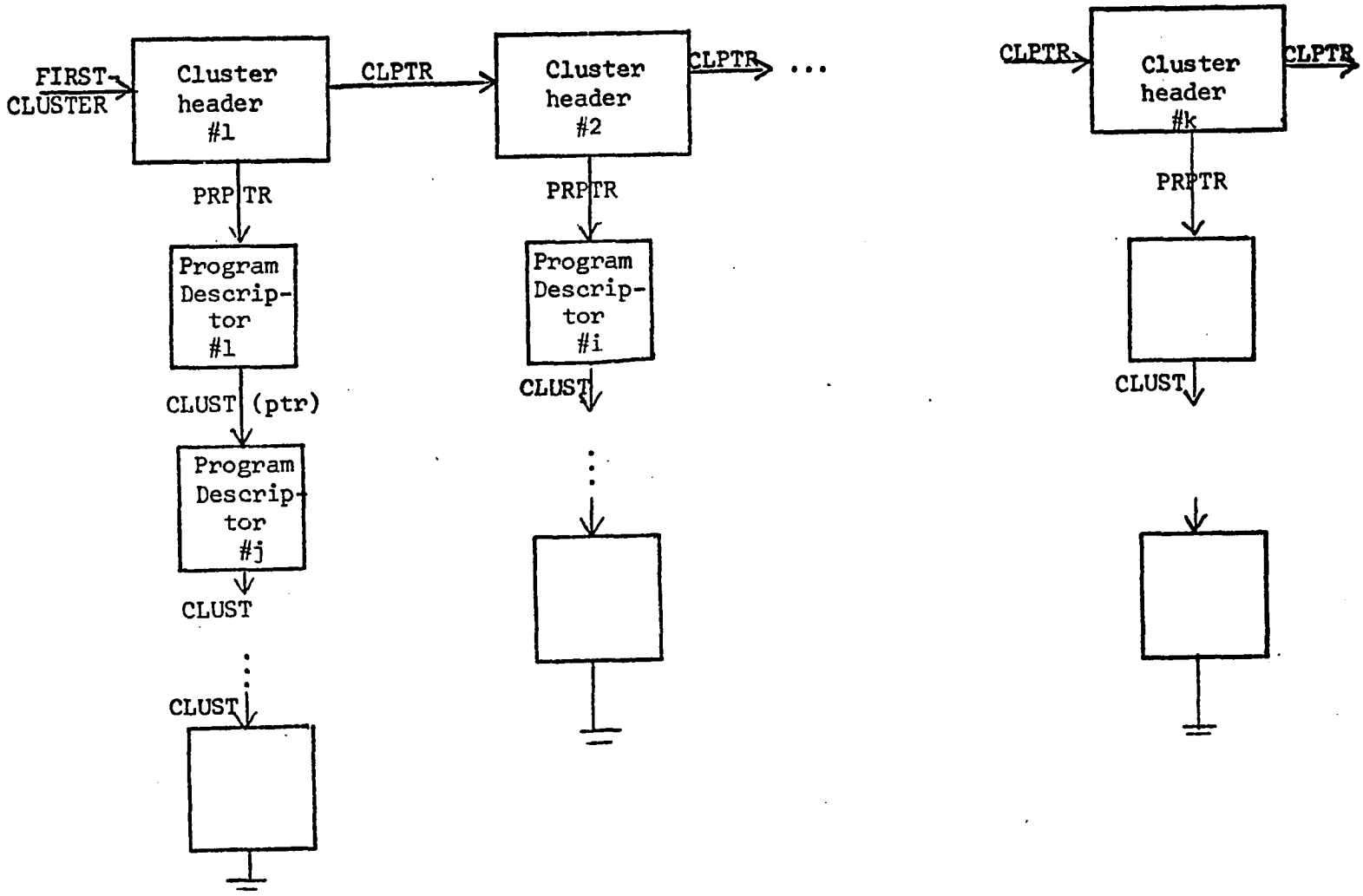
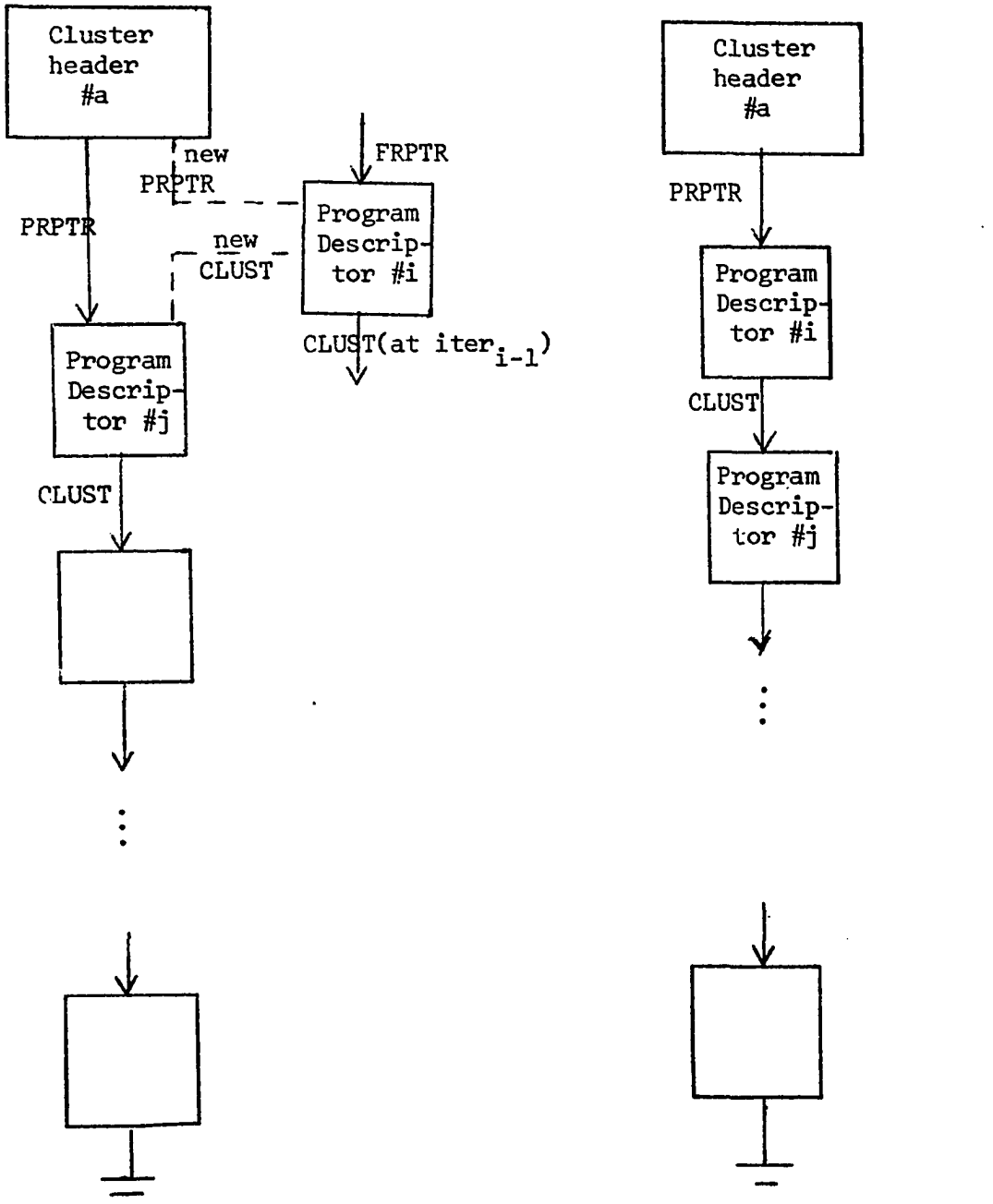


Figure A.2 Insertion of a Program into a Cluster



a. structure before inserting Program i

b. structure after inserting Program i

```
QUAD:  PROC;                                                          1
/*     QUADRATIC 0-1 PROGRAMMING          */                          2
      /*    DECLARATION DIVISION     */                               3
 DCL COST(1) FIXED BIN(31) CONTROLLED;                                4
 DCL PARTIAL(1,1) FIXED DEC(1) CONTROLLED;                            5
 DCL WORK(1) FIXED DEC(1) CONTROLLED;                                 6
 DCL (RESULT(1),SOLN(1)) FIXED DEC(1) CONTROLLED;                     7
 DCL (S(1),A(1),B(1)     ) FIXED BIN(31) CONTROLLED;                  8
 DCL (COSTMAX,NEW,MAX) FIXED BIN(31);                                 9
 DCL SUM FIXED BIN(31);                                              10
 DCL (TF,BB) FIXED DEC(1) INIT(1);                                   11
 DCL (KSUM,JSUM) FIXED BIN (31,0) STATIC;                            12
 DCL (JS,KS) FIXED BIN (31,0) STATIC;                                13
 NC=NPROG*NPROC;                                                     14
 KAMT=(NPROG-1)*(NPROG-2)/2;                                         15
 L=KAMT       +NPROG-1;                                              16
 ALLOCATE COST(L);                                                   17
 DO J=1 TO NPROG-1;                                                  18
 II=J+1;                                                             19
 DO I=II TO NPROG;                                                   20
 LL=(I-1)*(I-2)/2+J;                                                 21
 SUM=0;                                                              22
 DO K=1 TO NMSG;                                                     23
 SUM=SUM+TRANS(I,K)*TRANS(J,K)*MSGWT(K);                             24
 END;                                                                25
 COST(LL)=SUM;                                                       26
 END;                                                                27
 END;                                                                28
 /*    INITIAL ALLOCATION     */                                     29
 ALLOCATE  WORK(NO);                                                 30
 ALLOCATE RESULT(NPROG);                                             31
 NM=2*NPROC;                                                         32
 ALLOCATE S(NM);                                                     33
 ALLOCATE A(NPROG);                                                  34
 ALLOCATE B(NPROG);                                                  35
```

```
DO I=1 TO NPROG;
RESULT(I)=0;
A(I)=TRANS(I,NMSG+1);
B(I)=TRANS(I,NMSG+2);
END;
 JSUM=0;
 KSUM=0;
DO I=1 TO NM;
S(I)=BOUNDS(I);
 IF MOD(I,2)=0 THEN KSUM=KSUM + S(I);
      ELSE JSUM=JSUM+S(I);
END;
ALLOCATE PARTIAL(NO+1,NO);
IF INT_FACE='0'B THEN BEGIN;
  DO I=1 TO NO;
  PARTIAL(1,I)=0;
  END;
  PARTIAL(1,1)=2;
  NODE=1;
  END;
 ELSE BEGIN;
NODE=NPROG;
  KNT=0;
  DO I=1 TO NO;
KNT=KNT+ROOT(I)/2;
   DO N=1 TO NODE;
IF N>= KNT THEN PARTIAL(N,I)=ROOT(I);
    ELSE PARTIAL(N,I)=0;
  END;
 END;    END;
DO I=1 TO NO;
IF PARTIAL(NODE,I)=2 THEN DO;
  J=I/NPROG+1;
  K=I-(J-1)*NPROG;
  S(2*J-1)=S(2*J-1)-A(K);
  S(2*J)=S(2*J)-B(K);
```

```
    JSUM=JSUM-A(K);                                                    72
    KSUM=KSUM-B(K);                                                    73
    END;                                                               74
  END;                                                                 75
  /*     ALGORITHM        */                                          76
  KNTSOLN=1;                                                           77
  IF INT_FACE='0'B THEN                                                78
  MAX=0;  ELSE CALL OBJ(MAX);                                          79
  PUT SKIP LIST('MAX', MAX);                                           80
  COSTMAX=0;                                                           81
  MAXNODE=1;                                                           82
 /*    PARTITION STEP     */                                          83
  IF INT_FACE='1'B THEN GO TO SOLN_STK;                                84
 PART:                                                                 85
  DO I=1 TO NPROG;                                                     86
  IF A(I)>JSUM | B(I)> KSUM THEN GO TO  XEND;                          87
  DO J=1 TO NPROC;                                                     88
  NUM=(J-1)*NPROG+I;                                                   89
  IF A(I)>S(2*J-1) | B(I)>S(2*J)|PARTIAL(NODE,NUM) ¬=0                 90
       THEN GO TO INEND;                                               91
  DO K=1 TO NO;                                                        92
   PARTIAL(NODE+1,K)=PARTIAL(NODE,K);                                  93
  END;                                                                 94
  NODE=NODE+1;                                                         95
   PARTIAL(NODE,NUM)=2;                                                96
  IF NODE>MAXNODE THEN MAXNODE=NODE;                                   97
  GO TO CALC;                                                          98
INEND:  END;                                                           99
 XEND:  END;                                                          100
  GO TO FATHOM;                                                       101
 /*    CALCULATE BOUNDS     */                                        102
CALC:  S(2*J-1)=S(2*J-1)-A(I);                                         103
  S(2*J)=S(2*J)-B(I);                                                  104
  JSUM=JSUM-A(I);                                                      105
  KSUM=KSUM-B(I);                                                      106
  /*    TEST FOR A POSSIBLE SOLUTION      */                         107
```

```
 JS=0;                                                                    108
 KS=0;                                                                    109
 BB=0;                                                                    110
 DO I=1 TO NPROG;                                                         111
 RESULT(I)=0;                                                             112
 DO J=1 TO NPROC;                                                         113
 N=(J-1)*NPROG+I;                                                         114
 TF=PARTIAL(NODE,N)/2;                                                    115
 IF TF=1 THEN IF RESULT(I)=1 THEN GO TO FATHOM; ELSE RESULT(I)=1;         116
 END;                                                                     117
 IF RESULT(I)=0  THEN                                                     118
      DO;                                                                 119
      BB=1;                                                               120
      JS=JS+A(I);                                                         121
      KS=KS+B(I);                                                         122
      END;                                                                123
 END;                                                                     124
 IF KS> KSUM | JS> JSUM THEN GO TO FATHOM;                                125
 IF BB=1 THEN GO TO PART;                                                 126
 CALL OBJ(COSTMAX);                                                       127
 IF COSTMAX<MAX THEN GO TO PART;                                          128
 IF COSTMAX> MAX THEN DO;                                                 129
 DO WHILE(KNTSOLN>1);                                                     130
 KNTSOLN=KNTSOLN-1;                                                       131
   FREE SOLN;                                                             132
   END;                                                                   133
   MAX=COSTMAX;                                                           134
 END;                                                                     135
SOLN_STK:                                                                 136
 KNTSOLN=KNTSOLN+1;                                                       137
 ALLOCATE SOLN(NO);                                                       138
 DO I=1 TO NO;                                                            139
 SOLN(I)=PARTIAL(NODE,I)/2;                                               140
 END;                                                                     141
 PUT SKIP LIST('INSERT ON THE SOLN STACK');                               142
 PUT SKIP LIST('MAX',MAX);                                                143
```

```
  PUT SKIP EDIT(SOLN)(R(FMT));                                                  144
FMT:  FORMAT(COL(5),(NO)F(1));                                                  145
 /*  FATHOMING     */                                                          146
FATHOM:  IF NODE<=1 THEN GO TO TERM;                                            147
 /*   COMPARE THE LAST TWO ELEMENTS ON THE TREE STACK    */                     148
  DO I=1 TO NO;                                                                 149
  IF PARTIAL(NODE-1,I)¬=2 & PARTIAL(NODE,I) =2 THEN GO TO NEXT;                 150
  END;                                                                          151
 /*   BACKTRACKING     */                                                      152
  NODE=NODE-1;                                                                  153
  GO TO FATHOM;                                                                 154
NEXT:                                                                           155
  IND=I;                                                                        156
  PARTIAL(NODE,IND)=1;                                                          157
  N=(IND-1)/NPROG+1;                                                            158
  I=IND-(N-1)*NPROG;                                                            159
  S(2*N-1)=S(2*N-1)+A(I);                                                       160
  S(2*N)=S(2*N)+B(I);                                                           161
  KSUM=KSUM+B(I);                                                               162
  JSUM=JSUM+A(I);                                                               163
  GO TO PART;                                                                   164
  TERM:                                                                         165
 PUT SKIP LIST(' SOLUTION SET',MAX,MAX);                                        166
  DO WHILE(KNTSOLN>1);                                                          167
  PUT SKIP EDIT(SOLN)(R(FMT));                                                  168
  KNTSOLN=KNTSOLN-1;                                                            169
  FREE SOLN;                                                                    170
  END;                                                                          171
  PUT SKIP LIST('THE MAXIMUM NUMBER OF NODES ON THE TREE',MAXNODE);             172
  PUT SKIP LIST('NODES',NODE);                                                  173
/***************   PROCEDURES      ***********/                                174
 OBJ: PROC  (NEW);                                                              175
   DCL NEW FIXED BIN(31);                                                       176
   NEW=0;                                                                       177
  LX=NPROG-1;                                                                   178
  DO I=1 TO NO;                                                                 179
```

```
 WORK(I)=PARTIAL(NODE,I)/2;                                                    180
 END;                                                                          181
 DO I=1 TO NPROC;                                                              182
 NUM=(I-1)*NPROG;                                                              183
 DO J=1 TO LX;                                                                 184
 NJ=NUM+J;                                                                     185
 JJ=J+1;                                                                       186
 DO K=JJ TO NPROG;                                                             187
 NK=NUM+K;                                                                     188
 IF WORK(NJ)=1  &WORK(NK)=1    THEN N=1; ELSE  N=0;                            189
 LL=(K-1)*(K-2)/2+J;                                                           190
 NEW=NEW+N*COST(LL);                                                           191
 END;  END;  END;                                                              192
 RETURN;                                                                       193
 END;                                                                          194
 RETURN;                                                                       195
END;                                                                           196
```

```
/*****************************************************************/                1
/*                    CLUSTER ANALYSIS                           */                2
/*****************************************************************/                3
 DCL TRANS(1,1) CONTROLLED FIXED BIN;                                              4
  DCL BOUNDS(1) CONTROLLED FIXED BIN;                                              5
 DCL MSGWT(1) CONTROLLED;                                                          6
  DCL (NPROG,NMSG,NPROC,KNT,K2NT) FIXED BIN STATIC;                                7
   DCL I STATIC;                                                                   8
 NEXTIN:                                                                           9
   GET LIST (NPROG,NMSG,NPROC);                                                   10
PUT SKIP LIST(NPROG,NMSG,NPROC);                                                  11
   KNT=NMSG+2;                                                                    12
  DCL 1 PROGRAM BASED (P),                                                        13
     2 NUMB FIXED BIN,                                                            14
     2 CLUST POINTER,                                                             15
     2 FRPTR POINTER,                                                             16
     2 CLNO FIXED BIN,                                                            17
     2 NP FIXED BIN,                                                              18
     2 PROG (KNT  REFER (NP)) FLOAT;                                              19
  K2NT=2*KNT;                                                                     20
  DCL 1 CLUSTER BASED (CL),                                                       21
     2 CLPTR POINTER,                                                             22
     2 CLKNT FIXED BIN,                                                           23
     2 NUM FIXED BIN,                                                             24
     2 PRPTR POINTER,                                                             25
     2 NC FIXED BIN,                                                              26
     2 CENTER (K2NT REFER (NC)) FLOAT;                                            27
  DCL (MOVES,NO) FIXED BIN INIT(0);                                               28
  DCL (CREATE,INITIAL,UPDATE,NEWPROC) CHAR(7) STATIC;                             29
  DCL (TOTALDIST,D,DTEST) FLOAT      INIT(0);                                     30
   DCL MTHD(2) CHAR(7) INIT('FORGY  ','MACQUEN');                                 31
   DCL CEN_INIT(3) CHAR(7) INIT('FIRSTN ','BUSYPR ','BUSYMSG');                   32
   CREATE='INPUT  ';                                                              33
   NEWPROC='NO     ';                                                             34
  DCL (FIRST_CLUSTER,FIRST_PROG,LAST,C) POINTER STATIC;                           35
```

```
   N=2*NPROC;                                                                    36
   PUT SKIP LIST(N);                                                             37
   ALLOCATE TRANS(NPROG,KNT);                                                    38
   ALLOCATE BOUNDS(N);                                                           39
   ALLOCATE MSGWT(NMSG);                                                         40
/**********INITIAL PROGRAM,MESSAGE,PROCESSOR DESCRIPTION */                     41
   IF CREATE='INPUT  ' THEN CALL INPUT(NPROG,NMSG,NPROC);                        42
     ELSE BEGIN;                                                                 43
  DCL (ISEED,IY) FIXED BIN(31);                                                  44
   UNSPEC(ISEED)='00000000000000011111111111111111'B;                           45
   CALL GENER(ISEED,NPROG,NMSG,NPROC);                                          46
   PUT SKIP LIST ('OUT PUT OF THE TRANSITION MATRIX, BOUNDS, &MSGWT');          47
    PUT SKIP LIST(TRANS);                                                       48
   PUT SKIP LIST(BOUNDS,MSGWT);                                                 49
   END;                                                                         50
   LAST=NULL;                                                                   51
   DO I=1 TO NPROG;                                                             52
   ALLOCATE PROGRAM SET(P);                                                     53
   IF I=1 THEN FIRST_PROG=P;                                                    54
   ELSE LAST->FRPTR=P;                                                          55
   P->NUMB=I;                                                                   56
   P->CLNO=0;                                                                   57
   DO J=1 TO NMSG;                                                              58
   P->PROG(J)=TRANS(I,J)*MSGWT(J);                                              59
   END;                                                                         60
   P->PROG(KNT-1)=TRANS(I,NMSG+1);                                              61
   P->PROG(KNT)=TRANS(I,KNT);                                                   62
 PUT SKIP LIST(P->PROG);                                                        63
   LAST=P;                                                                      64
   END;                                                                         65
 INCEN:  P->FRPTR=NULL;                                                         66
   DO LC =1 TO 3;                                                               67
   INITIAL=CEN_INIT(LC);                                                        68
   DO LM=1 TO 2;                                                                69
   UPDATE=MTHD(LM);                                                             70
  PUT SKIP LIST(INITIAL,UPDATE);                                                71
```

```
/*************** INITIALIZE CENTROID                    */
   NO=0;
    KK=KNT+1;
   IF INITIAL='FIRSTN ' THEN CALL FIRSTN; ELSE
   IF INITIAL='BUSYPR ' THEN CALL BUSYPR;
     ELSE CALL BUSYMSG(TRANS);
   LAST=CL;
   IF UPDATE='FORGY   ' THEN MAX=10; ELSE MAX=4;
/************     BEGIN CLUSTER ANALYSIS          */
 FIRST:  LASTTOTAL=0;
 PUT SKIP LIST('TIME IN',TIME);
   DO ITIMES =1 TO MAX;
   P=FIRST_PROG;
   TOTALDIST=0;
   MOVES=0;
/*******     ASSIGN PROGRAMS TO CLUSTERS      */
   DO M=1 TO NPROG;
   CL=FIRST_CLUSTER;
   DO WHILE(CL¬=NULL &
            (P->PROG(KNT-1)+CL->CENTER(K2NT-1)> CL->CENTER(KNT-1) |
             P->PROG(KNT)+CL->CENTER(K2NT)> CL->CENTER(KNT)));
   CL=CL->CLPTR;
   END;
   IF CL = NULL           THEN GO TO UNABLE;
   IF P= NULL THEN GO TO OUTDO;
   D=DIST(P,CL);
   C=CL->CLPTR;
   DO WHILE (C¬=NULL);
   DO WHILE( (P->PROG(KNT-1)+C->CENTER(K2NT-1)> C->CENTER(KNT-1) |
             P->PROG(KNT)+C->CENTER(K2NT)> C->CENTER(KNT))& C¬=NULL);
   C=C->CLPTR;
   END;
   IF C=NULL THEN GO TO OUTDO;
   DTEST=DIST(P,C);
   IF D<=DTEST THEN GO TO NEXTCL;
   D=DTEST;
```

```
  CL=C;                                                                  108
NEXTCL: C=C->CLPTR;                                                      109
  END;                                                                   110
OUTDO:  IF CL = NULL THEN GO TO UNABLE;                                  111
  TOTALDIST=TOTALDIST+D;                                                 112
  IF CL->CLKNT¬=0 THEN P->CLUST=CL->PRPTR;                               113
    ELSE P->CLUST=NULL;                                                  114
  CL->CLKNT=CL->CLKNT+1;                                                 115
  CL->PRPTR=P;                                                           116
  IF P->CLNO¬=CL->NUM THEN BEGIN;                                        117
    MOVES=MOVES+1;                                                       118
  P->CLNO=CL->NUM;                                                       119
    END;                                                                 120
  DO J=KK TO K2NT;                                                       121
  CL->CENTER(J)=CL->CENTER(J)+P->PROG(J-KNT);                            122
  END;                                                                   123
  IF UPDATE='MACQUEN' THEN CALL UPDATEC;                                 124
  P=P->FRPTR;                                                            125
  END;                                                                   126
  IF MOVES=0|  ITIMES=MAX       THEN GO TO OUTPUT;                       127
  IF ABS(TOTALDIST-LASTTOTAL)<=.05 THEN GO TO OUTPUT;                    128
  IF UPDATE='FORGY  ' THEN CALL UPDATEC;                                 129
  CL=FIRST_CLUSTER;                                                      130
  DO L=1 TO NPROC;                                                       131
  CL->PRPTR=NULL;                                                        132
  DO J=KK TO K2NT;                                                       133
  CL->CENTER(J)=0;                                                       134
  END;                                                                   135
  CL->CLKNT=0;                                                           136
  CL=CL->CLPTR;                                                          137
  END;                                                                   138
  PUT SKIP LIST('TOTAL OF DISTANCES FROM CENTROIDS',TOTALDIST,           139
       'AT ITERATION',ITIMES);                                           140
 PUT SKIP LIST('MOVES',MOVES);                                           141
  LASTTOTAL=TOTALDIST;                                                   142
  END;                                                                   143
```

```
OUTPUT:  CL=FIRST_CLUSTER;                                                    144
  PUT SKIP LIST(ITIMES, TOTALDIST,LASTTOTAL);                                 145
PUT SKIP LIST('MOVES',MOVES);                                                 146
  ISEED=0;                                                                    147
  DO WHILE(CL¬=NULL);                                                         148
  PUT SKIP LIST('CLUSTER',CL->NUM,'NUMBER OF PROGRAMS',CL->CLKNT,             149
    'CENTROID & SUMS', CL->CENTER);                                           150
  P=CL->PRPTR;                                                                151
  C=P;                                                                        152
  MAX=CL->CLKNT;                                                              153
  PUT SKIP LIST('PROGRAMS');                                                  154
  DO WHILE(P¬=NULL);                                                          155
  PUT SKIP LIST(P->NUMB);                                                     156
  K=P->NUMB;                                                                  157
  LAST=C;                                                                     158
  DO I=1 TO MAX;                                                              159
  M=LAST->NUMB;                                                               160
  IF M>=K THEN GO TO JMP;                                                     161
  DO J=1 TO NMSG;                                                             162
  ISEED=ISEED+TRANS(K,J)*TRANS(M,J)*MSGWT(J);                                 163
  END;                                                                        164
JMP:  LAST=LAST->CLUST;                                                       165
  END;                                                                        166
  P->CLNO=0;                                                                  167
  P=P->CLUST;                                                                 168
  END;                                                                        169
  CL=CL->CLPTR;                                                               170
  END;                                                                        171
  PUT SKIP LIST('COST SAVED',ISEED);                                          172
  GO TO FINIS;                                                                173
UNABLE:  IF NEWPROC¬='YES     ' THEN BEGIN;                                   174
    PUT SKIP LIST('UNABLE TO COMPLETE');                                      175
    GO TO OUTPUT;                                                             176
    END;                                                                      177
  IF NPROC=N THEN GO TO OUTPUT;                                               178
  I=KNT;                                                                      179
```

```
    TOTALDIST=0;                                                          180
    CALL ALLOCTE(LAST);                                                   181
    NPROC=NPROC+1;                                                        182
    CL->CENTER(KNT-1)=FIRST_CLUSTER->CENTER(KNT-1);                       183
    CL->CENTER(KNT)=FIRST_CLUSTER->CENTER(KNT);                           184
    LAST->CLPTR=CL;                                                       185
    LAST=CL;                                                              186
    CL=FIRST_CLUSTER;                                                     187
    DO K=1 TO NPROC;                                                      188
    CL->CLKNT=0;                                                          189
    P=CL->PRPTR;                                                          190
    DO WHILE(P¬=NULL);                                                    191
    C=P->CLUST;                                                           192
    P->CLUST=NULL;                                                        193
    P=C;                                                                  194
    END;                                                                  195
    DO I=KK TO K2NT;                                                      196
    CL->CENTER(I)=0;                                                      197
    END;                                                                  198
    CL=CL->CLPTR;                                                         199
    END;                                                                  200
    GO TO FIRST;                                                          201
/*******   PROCEDURES  FOR PROGRAM,MESSAGE,PROCESSOR INITIALIZATION */   202
   (NOCONVERSION):                                                        203
  INPUT:  PROCEDURE(NPROG,NMSG,NPROC);                                    204
   DCL (NPROG,NMSG,NPROC) FIXED BIN;                                      205
    PUT SKIP LIST ('OUT PUT OF THE TRANSITION MATRIX, BOUNDS, &MSGWT');   206
    DO I= 1 TO NPROG;                                                     207
    GET EDIT((TRANS(I,J)  DO J=1 TO KNT ))(R(FMT));                       208
    END;                                                                  209
   PUT SKIP LIST(TRANS);                                                  210
    N=2*NPROC;                                                            211
    GET EDIT((BOUNDS(I) DO I=1 TO N))(R(FMT));                            212
    PUT SKIP LIST(BOUNDS);                                                213
    GET EDIT((MSGWT(I) DO I=1 TO NMSG))(R(FMT));                          214
   PUT SKIP LIST(MSGWT);                                                  215
```

```
 FMT:  FORMAT( COL(1),(20)F(4));                                    216
  RETURN;                                                           217
  END;                                                              218
GENER:  PROCEDURE(ISEED,NPROG,NMSG,NPROC);                          219
 DCL (NPROG,NMSG,NPROC) FIXED BIN;                                  220
 DCL (ISEED,IY) FIXED BIN(31);                                      221
 DCL Y FLOAT BIN;                                                   222
 KNT=NMSG+2;                                                        223
  DO I=1 TO NPROG;                                                  224
 DO J=1 TO NMSG;                                                    225
  CALL RANDU(ISEED,IY,Y);                                           226
  ISEED=IY;                                                         227
  TRANS(I,J)=FUNCT1(Y);                                             228
  END;                                                              229
  CALL RANDU(ISEED,IY,Y);                                           230
  ISEED=IY;                                                         231
  TRANS(I,NMSG+1)=FUNCT2(Y);                                        232
  CALL RANDU(IY,ISEED,Y);                                           233
  TRANS(I,KNT)=FUNCT2(Y);                                           234
  END;                                                              235
  N=2*NPROC;                                                        236
  DO I=1 TO N;                                                      237
  CALL RANDU(ISEED,IY,Y);                                           238
  ISEED=IY;                                                         239
  BOUNDS(I)=FUNCT3(Y);                                              240
  END;                                                              241
  DO I=1 TO NMSG;                                                   242
  CALL RANDU(ISEED,IY,Y);                                           243
  ISEED=IY;                                                         244
  MSGWT(I)=FUNCT4(Y);                                               245
  END;                                                              246
  RETURN;                                                           247
  END;                                                              248
/* ********** PROCEDURES FOR CENTROID INITIALIZATION  */            249
 FIRSTN:  PROCEDURE;                                                250
 PUT SKIP LIST('TIME IN',TIME);                                     251
```

```
    LAST=NULL;                                                          252
    P=FIRST_PROG;                                                       253
    DO I=1 TO NPROC;                                                    254
    CALL ALLOCTE(LAST);                                                 255
    DO J=1 TO NMSG;                                                     256
    CL->CENTER(J)=P->PROG(J);                                           257
    END;                                                                258
    IF I=1 THEN FIRST_CLUSTER=CL;                                       259
       ELSE LAST->CLPTR=CL;                                             260
    LAST=CL;                                                            261
    P=P->FRPTR;                                                         262
    END;                                                                263
   PUT SKIP LIST('TIME OUT',TIME);                                      264
    RETURN;                                                             265
    END;                                                                266
 ALLOCTE:   PROCEDURE(LAST);                                            267
  DCL LAST POINTER;                                                     268
    ALLOCATE CLUSTER SET(CL);                                           269
    CL->CLKNT=0;                                                        270
    CL->CLPTR=NULL;                                                     271
    NO=NO+1;                                                            272
    CL->NUM=NO;                                                         273
    CL->PRPTR=NULL;                                                     274
    DO J=1 TO K2NT;                                                     275
    CL->CENTER(J)=0;                                                    276
    END;                                                                277
    IF NPROC>N/2 THEN RETURN;                                           278
    CL->CENTER(KNT-1)=BOUNDS(2*I-1);                                    279
    CL->CENTER(KNT)=BOUNDS(2*I);                                        280
    RETURN;                                                             281
    END;                                                                282
BUSYPR:  PROCEDURE;                                                     283
    DCL SUM(1) CONTROLLED FLOAT;                                        284
 PUT SKIP LIST('TIME IN',TIME);                                         285
    ALLOCATE SUM(NPROG);                                                286
    DCL S FLOAT;                                                        287
```

```
   DCL TEMP POINTER;                                                  288
   LAST=NULL;                                                         289
   P=FIRST_PROG;                                                      290
   DO I=1 TO NPROG;                                                   291
   S=0;                                                               292
   DO J=1 TO NMSG;                                                    293
   S=S+P->PROG(J);                                                    294
   END;                                                               295
   SUM(I)=S;                                                          296
   P=P->FRPTR;                                                        297
   END;                                                               298
   DO I=1 TO NPROC;                                                   299
   CALL ALLOCTE(LAST);                                                300
   IF I=1 THEN FIRST_CLUSTER=CL;                                      301
        ELSE LAST->CLPTR=CL;                                          302
   TEMP=FIRST_PROG;                                                   303
   S=SUM(1);                                                          304
   K=1;                                                               305
   LAST=CL;                                                           306
   P=FIRST_PROG;                                                      307
   DO J=1 TO NPROG;                                                   308
   IF S>= SUM(J) THEN GO TO DOEND;                                    309
   S=SUM(J);                                                          310
   K=J;                                                               311
   TEMP=P;                                                            312
DOEND: P=P->FRPTR;                                                    313
   END;                                                               314
   DO J=1 TO NMSG;                                                    315
   CL->CENTER(J)=TEMP->PROG(J);                                       316
   END;                                                               317
   SUM(K)=0;                                                          318
   END;                                                               319
   FREE SUM;                                                          320
 PUT SKIP LIST('TIME OUT',TIME);                                      321
   RETURN;                                                            322
   END;                                                               323
```

```
BUSYMSG:  PROCEDURE(TRANS);                                          324
  DCL TRANS(*,*) FIXED BIN(15);                                      325
DCL SUM(1) CONTROLLED FIXED BIN(31);                                 326
 DCL TEMP POINTER;                                                   327
  DCL S FIXED BIN(31);                                               328
PUT SKIP LIST('TIME IN',TIME);                                       329
  ALLOCATE SUM(NMSG);                                                330
  DO J=1 TO NMSG;                                                    331
 S=0;                                                                332
  DO I=1 TO NPROG;                                                   333
  S=S+TRANS(I,J);                                                    334
  END;                                                               335
  SUM(J)=S*MSGWT(J);                                                 336
  END;                                                               337
  TEMP=NULL;                                                         338
  DO K=1 TO NPROC;                                                   339
  L=1;                                                               340
  S=SUM(1);                                                          341
  DO I=1 TO NMSG;                                                    342
  IF S>= SUM(I) THEN GO TO ENDDO;                                    343
  L=I;                                                               344
  S=SUM(I);                                                          345
ENDDO:  END;                                                         346
  I=K;                                                               347
  CALL ALLOCTE(TEMP);                                                348
  IF K=1 THEN FIRST_CLUSTER=CL;                                      349
       ELSE TEMP->CLPTR=CL;                                          350
  TEMP=CL;                                                           351
  CL->CENTER(L)=MSGWT(L);                                            352
  SUM(L)=0;                                                          353
  END;                                                               354
 PUT SKIP LIST('TIME OUT',TIME);                                     355
  RETURN;                                                            356
  END;                                                               357
UPDATEC:  PROCEDURE;                                                 358
  IF UPDATE='FORGY  ' THEN CL=FIRST_CLUSTER;                         359
```

```
OVER:   IF CL=NULL THEN RETURN;                                                  360
  DO I=1 TO NMSG;                                                                361
  IF CL->CLKNT¬=0 THEN CL->CENTER(I)=CL->CENTER(KNT+I)/CL->CLKNT;                362
    ELSE CL->CENTER(I)=0;                                                        363
  END;                                                                           364
  IF UPDATE='MACQUEN' THEN RETURN;                                               365
      CL->CLKNT=0;                                                               366
  CL->CENTER(K2NT-1)=0;                                                          367
  CL->CENTER(K2NT)=0;                                                            368
CL=CL->CLPTR;                                                                    369
  GO TO OVER;                                                                    370
  RETURN;                                                                        371
  END;                                                                           372
  GO TO NEXTIN;  END;                                                            373
```

```
/*****************************************************************/                1
/*                      HEURISTIC   PROCEDURE                    */                2
/*****************************************************************/                3
   DCL (A,B)  FLOAT;                                                               4
 DCL TRANS(1,1) CONTROLLED FIXED BIN;                                              5
 DCL MSGWT(1) CONTROLLED;                                                          6
  DCL (NPROG,NMSG,NPROC,KNT,K2NT) FIXED BIN STATIC;                                7
   DCL I STATIC;                                                                   8
   GET LIST(NPROG,NMSG);                                                           9
   PUT SKIP LIST(NPROG,NMSG);                                                     10
   KNT=NMSG+2;                                                                    11
  DCL 1 PROGRAM BASED (P),                                                        12
     2 NUMB FIXED BIN,                                                            13
     2 CLUST POINTER,                                                             14
     2 FRPTR POINTER,                                                             15
     2 CLNO FIXED BIN,                                                            16
     2 NP.FIXED BIN,                                                              17
     2 PROG (KNT   REFER (NP)) FLOAT;                                             18
   K2NT=KNT;                                                                      19
  DCL 1 CLUSTER BASED (CL),                                                       20
     2 CLPTR POINTER,                                                             21
     2 CLKNT FIXED BIN,                                                           22
     2 NUM FIXED BIN,                                                             23
     2 PRPTR POINTER,                                                             24
   2 LSTPTR POINTER,                                                              25
     2 NC FIXED BIN,                                                              26
     2 CENTER (K2NT REFER (NC)) FLOAT;                                            27
   DCL CREATE CHAR(7);                                                            28
   GET LIST(CREATE);                                                              29
   PUT SKIP LIST(CREATE);                                                         30
  DCL (TOTALDIST,D,DTEST) FLOAT     INIT(0);                                     31
  DCL (FIRST_CLUSTER,FIRST_PROG,LAST,C) POINTER STATIC;                          32
   ALLOCATE TRANS(NPROG,KNT);                                                     33
   ALLOCATE MSGWT(NMSG);                                                          34
/**********INITIAL PROGRAM,MESSAGE,PROCESSOR DESCRIPTION */                      35
```

```
   IF CREATE='INPUT  ' THEN CALL INPUT(NPROG,NMSG,NPROC);                    36
     ELSE BEGIN;                                                             37
  DCL (ISEED,IY) FIXED BIN(31);                                              38
   UNSPEC(ISEED)='00000000000000011111111111111111'B;                        39
   CALL GENER(ISEED,NPROG,NMSG,NPROC);                                       40
   PUT SKIP LIST ('OUT PUT OF THE TRANSITION MATRIX, BOUNDS, &MSGWT');       41
   PUT SKIP LIST(A,B    ,MSGWT);                                             42
    PUT SKIP LIST(TRANS);                                                    43
   END;                                                                      44
   LAST=NULL;                                                                45
   DO I=1 TO NPROG;                                                          46
   ALLOCATE PROGRAM SET(P);                                                  47
   IF I=1 THEN FIRST_PROG=P;                                                 48
   ELSE LAST->FRPTR=P;                                                       49
   P->NUMB=I;                                                                50
   P->CLNO=0;                                                                51
   DO J=1 TO NMSG;                                                           52
   P->PROG(J)=TRANS(I,J)*MSGWT(J);                                           53
   END;                                                                      54
   P->PROG(KNT-1)=TRANS(I,NMSG+1);                                           55
   P->PROG(KNT)=TRANS(I,KNT);                                                56
 PUT SKIP LIST(P->PROG);                                                     57
   LAST=P;                                                                   58
   END;                                                                      59
 INCEN:  P->FRPTR=NULL;                                                      60
/*************** INITIALIZE CENTROID              */                        61
   NPROC=NPROG;                                                              62
   CALL FIRSTN;                                                              63
   CL=FIRST_CLUSTER;                                                         64
   DO I=1 TO NPROC;                                                          65
   PUT SKIP LIST( CL->CENTER);                                               66
   CL=CL->CLPTR;                                                             67
   END;                                                                      68
   FREE TRANS;                                                               69
  FREE MSGWT;                                                                70
  DCL LASTTOTAL FLOAT;                                                       71
```

```
  DCL (DLAST,DPOINTER      ) POINTER STATIC;                          72
  /*   ASSIGNMENT      */                                            73
  LASTTOTAL=0;                                                       74
ITERATE:  CL=FIRST_CLUSTER;                                          75
  LAST=NULL ;                                                        76
  TOTAL=LASTTOTAL;                                                   77
  DO WHILE(CL¬=NULL);                                                78
  C=FIRST_CLUSTER;                                                   79
  NO=0;                                                              80
  CALL DCUNTIL(C);                                                   81
  IF C=NULL THEN GO TO ENDDO;                                        82
  D=DISTCL(CL,C);                                                    83
  DLAST=LAST;                                                        84
  DPOINTER=C;                                                        85
PUT SKIP LIST('INIT',D,DLAST->NUM,DPOINTER->NUM);                    86
  LAST=C;                                                            87
  C=C->CLPTR;                                                        88
  DO WHILE(C¬=NULL);                                                 89
  CALL DCUNTIL(C);                                                   90
  IF C=NULL THEN GO TO  MERGE;                                       91
  DTEST=DISTCL(CL,C);                                                92
  IF DTEST>=D THEN GO TO DOEND;                                      93
  D=DTEST;                                                           94
  DLAST=LAST;                                                        95
  DPOINTER=C;                                                        96
PUT SKIP LIST('LOOP',D,DLAST->NUM,DPOINTER->NUM);                    97
DOEND: LAST=C;                                                       98
       C=C->CLPTR;                                                   99
  END;                                                              100
 /*    MERGE      */                                               101
MERGE:  NO=NO+1;                                                    102
  PUT SKIP LIST(CL->NUM,DPOINTER->NUM);                             103
IF DLAST->CLPTR¬=DPOINTER THEN GO TO  FINIS;                        104
 DLAST->CLPTP=DPOINTER->CLPTR;                                      105
  CL->CLKNT=CL->CLKNT+DPOINTER->CLKNT;                              106
  C=CL->LSTPTR;                                                     107
```

```
   C->CLUST=DPOINTER->PRPTR;                                                  108
   CL->LSTPTR=DPOINTER->LSTPTR;                                               109
   DO K=1 TO NMSG;                                                            110
   CL->CENTER(K)=MAX(CL->CENTER(K),DPOINTER->CENTER(K));                      111
   END;                                                                       112
   CL->CENTER(KNT-1)=CL->CENTER(KNT-1)+DPOINTER->CENTER(KNT-1);               113
   CL->CENTER( KNT )=CL->CENTER( KNT )+DPOINTER->CENTER( KNT );               114
   FREE DPOINTER->CLUSTER;                                                    115
   NPROC=NPROC-1;                                                             116
   TOTAL=TOTAL+D;                                                             117
   PUT SKIP LIST(CL->NUM);                                                    118
   ENDDO:  CL=CL->CLPTR;                                                      119
   END;                                                                       120
   PUT SKIP LIST(NPROC,TOTAL,LASTTOTAL);                                      121
   IF NO¬=0 THEN GO TO ITERATE;                                               122
OUTPUT:  CL=FIRST_CLUSTER;                                                    123
   DO WHILE(CL¬=NULL);                                                        124
   PUT SKIP LIST('CLUSTER',CL->NUM,'NUMBER OF PROGRAMS',CL->CLKNT,            125
     'CENTROID & SUMS', CL->CENTER);                                          126
   P=CL->PRPTR;                                                               127
   DO WHILE(P¬=NULL);                                                         128
   PUT SKIP LIST('PROGRAMS',P->NUMB,'MSG & SNDS',P->PROG);                    129
   P=P->CLUST;                                                                130
   END;                                                                       131
   CL=CL->CLPTR;                                                              132
   END;                                                                       133
/*******   PROCEDURES  FOR PROGRAM,MESSAGE,PROCESSOR INITIALIZATION */       134
   DOUNTIL:  PROC(C);                                                         135
   DCL  C        POINTER;                                                     136
   DO WHILE(C¬=NULL);                                                         137
        IF   C->CENTER(KNT-1)+CL->CENTER(KNT-1)>A |                           138
             C->CENTER( KNT )+CL->CENTER( KNT )>B | C =CL                     139
     THEN BEGIN;                                                              140
   LAST=C;                                                                    141
   C=C->CLPTR;                                                                142
   END;                                                                       143
```

```
  ELSE RETURN;                                                                  144
  END;                                                                          145
  RETURN;                                                                       146
  END;                                                                          147
 (NOCONVERSION):                                                                148
INPUT:  PROCEDURE(NPROG,NMSG,NPROC);                                            149
 DCL (NPROG,NMSG,NPROC) FIXED BIN;                                              150
  PUT SKIP LIST ('OUT PUT OF THE TRANSITION MATRIX, BOUNDS, &MSGWT');           151
  DO I= 1 TO NPROG;                                                             152
  GET EDIT((TRANS(I,J)  DO J=1 TO KNT ))(R(FMT));                               153
  END;                                                                          154
 PUT SKIP LIST(TRANS);                                                          155
  GET EDIT(A,B)(R(FMT));                                                        156
  PUT SKIP LIST(A,B);                                                           157
  GET EDIT((MSGWT(I) DO I=1 TO NMSG))(R(FMT));                                  158
 PUT SKIP LIST(MSGWT);                                                          159
 FMT:  FORMAT( COL(1),(20)F(4));                                                160
  RETURN;                                                                       161
  END;                                                                          162
GENER:  PROCEDURE(ISEED,NPROG,NMSG,NPROC);                                      163
 DCL (NPROG,NMSG,NPROC) FIXED BIN;                                              164
 DCL (ISEED,IY) FIXED BIN(31);                                                  165
 DCL Y FLOAT BIN;                                                               166
 KNT=NMSG+2;                                                                    167
  DO I=1 TO NPROG;                                                              168
 DO J=1 TO NMSG;                                                                169
  CALL RANDU(ISEED,IY,Y);                                                       170
  ISEED=IY;                                                                     171
  TRANS(I,J)=FUNCT1(Y);                                                         172
  END;                                                                          173
  CALL RANDU(ISEED,IY,Y);                                                       174
  ISEED=IY;                                                                     175
  TRANS(I,NMSG+1)=FUNCT2(Y);                                                    176
  CALL RANDU(IY,ISEED,Y);                                                       177
  TRANS(I,KNT)=FUNCT2(Y);                                                       178
  END;                                                                          179
```

```
   CALL RANDU(ISEED,IY,Y);                                                  180
   ISEED=IY;                                                                181
   A=FUNCT3(Y);    B=    FUNCT3(Y)+4;                                       182
   DO I=1 TO NMSG;                                                          183
   CALL RANDU(ISEED,IY,Y);                                                  184
   ISEED=IY;                                                                185
   MSGWT(I)=FUNCT4(Y);                                                      186
   END;                                                                     187
   RETURN;                                                                  188
   END;                                                                     189
/* ************ PROCEDURES FOR CENTROID INITIALIZATION  */                  190
 FIRSTN:  PROCEDURE;                                                        191
   LAST=NULL;                                                               192
   P=FIRST_PROG;                                                            193
   DO I=1 TO NPROC;                                                         194
   ALLOCATE CLUSTER SET(CL);                                                195
   CL->CLKNT=1;                                                             196
   CL->PRETR=P;                                                             197
   CL->LSTPTR=P;                                                            198
   P->CLUST=NULL;                                                           199
   CL->NUM=I;                                                               200
   DO J=1 TO NMSG;                                                          201
   CL->CENTER(J)=P->PROG(J);                                                202
   END;                                                                     203
   CL->CENTER(KNT-1)=P->PROG(KNT-1);                                        204
   CL->CENTER(KNT)=P->PROG(KNT);                                            205
   IF I=1 THEN FIRST_CLUSTER=CL;                                            206
      ELSE LAST->CLPTR=CL;                                                  207
   LAST=CL;                                                                 208
   P=P->FRPTR;                                                              209
   END;                                                                     210
   CL->CLPTR=NULL;                                                          211
   RETURN;                                                                  212
   END;                                                                     213
   FINIS:  END;                                                             214
```

```
/********FUNCTION PROCEDURES        */                                    1
DISTCL: PROCEDURE(P,CL);                                                  2
  DCL (P,CL) POINTER;                                                     3
  DCL(NUMR,DENOM) FLOAT;                                                  4
   NUMR=0;                                                                5
   DENOM=0;                                                               6
   DO I=1 TO NMSG;                                                        7
   NUMR=NUMR+ABS(P->CENTER(I)-CL->CENTER(I));                             8
   DENOM=DENOM+P->CENTER(I)+CL->CENTER(I);                                9
  END;                                                                   10
  IF DENOM=0 THEN DENOM=.00005;                                          11
   RETURN(NUMR/DENOM);                                                   12
   END;                                                                  13
 FUNCT1:  PROCEDURE(Y);                                                  14
   DCL Y FLOAT BIN;                                                      15
   RETURN(Y+.55);                                                        16
   END;                                                                  17
 FUNCT2:  PROCEDURE(Y);                                                  18
   DCL Y FLOAT BIN;                                                      19
   RETURN(10*Y+1);                                                       20
   END;                                                                  21
 FUNCT3:  PROCEDURE(Y);                                                  22
   DCL Y FLOAT BIN;                                                      23
   RETURN(25*Y+1);                                                       24
   END;                                                                  25
 FUNCT4:  PROCEDURE(Y);                                                  26
   DCL Y FLOAT BIN;                                                      27
   RETURN(5*Y+1);                                                        28
   END;                                                                  29
* PROCESS ;                                                              30
    (NOFIXEDOVERFLOW);                                                   31
   RANDU: PROC(IX,IL,Y);                                                 32
    DCL (KX,KY) FIXED BIN (31,0);                                        33
    DCL (IX,IL) FIXED BIN(31,0);                                         34
       DCL Y FLOAT;                                                      35
```

```
DCL          YY INIT (0.4656613E-9);                                    36
      UNSPEC(KX)='01010010100011000011011100010101'B;                   37
       UNSPEC(KY)='01111111111111111111111111111111'B;                  38
      IL=IX*KX;                                                         39
IF  IL<=0 THEN IL=KY+IL+1;                                              40
     Y=IL;    Y=Y*YY;                                                   41
     RETURN;                                                            42
    END;                                                                43
```

APPENDIX B

Models used in the programs in Appendix A

Note that the /* convention is used to improve readability. Lines containing the comments are not part of the data.

```
/*****************************************************************/
/*                        USED IN SIMULATION                     */
/*                n = 10          k = 10     m = 6               */
/*****************************************************************/
/*                      SIMULATION PARAMETERS
        'APERIOD '          100              10          5
 ------------------------------------------------------------------
                          COLUMN NUMBERS
          1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
/*                  PROGRAM-MESSAGE TRANSITION TABLE              */

  1   1   0   1   0   0   0   0   0   1   4  25
  0   0   1   0   1   1   0   0   1   0   6  35
  0   0   0   0   1   1   0   0   1   1   5  30
  0   0   0   1   1   1   0   0   1   0   5  35
  1   1   0   1   0   0   0   0   0   0   7  50
  1   1   0   1   0   0   1   0   0   0   5  25
  0   0   1   0   0   0   1   1   0   1   7  40
  0   1   0   0   0   0   1   1   0   1   9  60
  0   0   1   1   1   0   1   0   1   0  10  45
  0   0   1   1   1   0   0   1   1   0   6  55

/*                             BOUNDS                             */
  16 100  16 100  16 100  16 100  16 100  16 100

/*                          MESSAGE WEIGHTS                       */
   5   5  10   5   7   7  10  10   5  10
```

```
/*****************************************************************/
/*                                                               */
/*                            LARGE                              */
/*                n = 10        k = 10      m = 5                */
/*****************************************************************/
/*                         COLUMN NUMBERS                        */
          1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
/*                  PROGRAM-MESSAGE TRANSITION TABLE             */
  1   0   1   0   1   0   0   0   0   0   6   9
  0   0   1   0   1   0   1   0   1   1   6   8
  0   1   0   1   1   1   1   0   0   1   7   7
  0   1   0   0   0   1   1   1   1   0   8  10
  0   1   0   0   1   1   0   0   1   0   7   6
  0   1   0   0   0   0   1   0   0   0   8   8
  1   0   1   1   0   0   0   1   1   1   6   6
  1   1   0   1   1   0   0   0   0   0   7   8
  0   0   0   1   0   0   1   1   0   1   6   7
  1   0   1   1   1   1   0   1   0   0   8   9

/*                           BOUNDS                              */
 20  50   20  50   20  50   20  50   20  50   20  50
/*                       MESSAGE WEIGHTS                         */
  1   1   2   1   3   1   3   2   2   2
```

```
/*****************************************************************/
/*                          MIXED                                */
/*              n = 50      k = 10      m = 5                    */
/*****************************************************************/
/*                       COLUMN NUMBERS                          */
         1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
/*                  PROGRAM-MESSAGE TRANSITION TABLE             */
  1   1   1   1   1   1   1   0   0   0   6  12
  0   0   0   1   0   0   0   1   0   0   1   2
  1   0   0   0   0   1   0   0   0   0   2   1
  0   0   1   0   0   0   0   0   1   0   1   2
  0   0   0   0   0   0   0   0   1   1   1   3
  0   0   0   1   0   0   0   1   0   0   2   1
  0   0   0   1   0   0   0   1   0   0   1   3
  0   0   0   1   0   1   0   0   0   0   1   3
  0   1   0   0   0   1   0   0   0   0   1   3
  0   0   0   0   0   1   1   0   0   0   1   1
  0   0   0   0   1   0   0   1   0   0   2   4
  0   0   0   1   0   0   0   0   1   0   1   1
  0   0   0   1   0   0   0   1   0   0   1   1
  0   0   0   1   0   0   0   1   0   0   1   1
  0   0   0   0   1   0   1   0   0   0   1   2
  0   0   0   1   0   1   0   0   0   0   2   2
  0   1   0   0   0   0   1   0   0   0   2   2
  1   0   0   0   0   0   0   0   0   0   2   1
  0   0   0   0   1   0   0   0   1   0   2   3
  0   0   0   0   0   1   1   0   0   0   1   1
  0   1   0   0   0   0   0   0   0   1   1   2
  1   0   0   0   1   1   0   0   0   0   2   4
  0   0   0   0   0   1   1   0   0   0   1   1
  0   0   0   1   0   0   1   0   0   0   1   2
  0   0   0   1   0   0   0   1   0   0   2   3
```

(program continues next page)

```
 0   0   0   1   1   0   0   0   0   0   1   3
 0   0   0   0   1   0   0   0   1   0   2   4
 0   0   0   0   1   0   0   0   1   0   2   4
 0   0   0   1   0   0   0   0   1   0   2   4
 0   0   0   0   1   0   0   0   1   0   2   1
 1   0   1   1   0   0   0   0   0   0   4   9
 0   0   0   0   0   1   1   0   1   0   4   8
 0   0   0   0   1   0   0   1   0   0   1   3
 1   1   0   0   0   0   1   0   0   0   2   3
 0   0   0   0   1   1   1   0   0   0   1   1
 0   0   0   0   1   0   0   0   1   0   1   3
 0   0   0   1   0   0   0   1   0   0   1   1
 0   0   1   1   0   0   0   0   0   0   2   2
 0   0   0   0   1   0   0   1   0   0   1   2
 0   0   0   0   1   0   1   0   1   0   1   2
 1   0   1   0   1   0   0   0   0   0   3  12
 0   0   0   0   0   0   1   0   0   1   1   4
 1   0   0   1   0   0   0   0   0   0   2   3
 0   0   0   0   1   1   0   1   0   0   2   2
 0   0   0   1   0   0   0   0   1   0   1   2
 0   0   0   0   0   1   0   1   0   0   1   2
 0   0   0   0   1   0   0   1   0   1   2   3
 0   0   0   0   1   1   0   0   0   0   1   2
 0   0   1   0   0   0   0   0   1   0   2   2
 0   0   1   1   1   1   1   1   0   0   6  10
/*                      BOUNDS                                    */
  20  50  20  50  20  50  20  50  20  50  20  50
/*                   MESSAGE WEIGHTS                              */
  1   1   2   1   3   1   3   2   2   2
```

```
/*****************************************************************************/
/*                                                                           */
/*                                  SMALL                                    */
/*                      n = 50       k = 10       m = 5                       */
/*****************************************************************************/
/*                              COLUMN NUMBERS                               */
        1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
/*                    PROGRAM-MESSAGE TRANSITION TABLE                       */
  1   0   1   0   0   0   0   0   1   0   1   6
  1   1   0   0   0   0   0   0   0   0   2   5
  1   0   0   1   0   0   0   0   0   0   2   5
  1   0   0   1   0   0   0   0   0   0   3   4
  0   1   0   0   1   0   0   0   0   0   3   3
  0   0   1   0   0   0   0   0   0   1   2   2
  0   0   1   0   1   0   0   0   0   0   1   1
  0   0   0   0   0   0   1   0   0   1   1   6
  0   0   0   1   0   0   1   0   0   0   1   5
  0   0   0   0   1   1   0   0   0   0   1   5
  0   0   1   0   0   0   0   1   0   1   2   4
  0   0   0   0   0   1   1   0   0   0   2   3
  0   0   0   1   1   0   0   0   0   0   2   3
  1   0   0   0   0   1   0   0   0   0   3   2
  0   0   0   0   0   1   0   0   0   1   2   1
  0   1   0   0   0   0   0   1   0   0   2   6
  0   0   0   1   0   1   0   0   0   0   2   5
  1   0   0   0   0   0   0   0   1   0   2   5
  1   0   0   0   0   0   1   0   0   0   1   4
  0   0   0   1   0   0   0   0   1   0   1   3
  0   0   1   0   1   0   0   0   0   0   1   2
  0   0   1   0   0   1   0   0   0   0   3   1
  0   0   0   0   1   0   0   0   1   0   1   5
  0   0   0   0   1   0   0   0   1   0   1   5
  0   0   0   1   1   0   0   0   0   0   2   4
```

(program continues next page)

```
   0   0   0   0   1   1   0   0   0   0   1   3
   0   0   0   0   1   0   0   1   0   0   2   2
   0   0   0   0   1   0   0   1   0   0   3   4
   0   0   0   0   0   0   1   0   0   1   1   3
   0   1   0   0   0   0   1   0   0   0   3   2
   0   0   0   0   0   0   0   1   0   1   2   4
   0   0   0   1   1   0   0   0   0   0   2   3
   0   0   0   1   0   0   0   1   0   0   2   4
   0   0   0   0   1   0   0   1   0   0   2   3
   0   1   0   1   0   0   0   0   0   0   1   5
   0   0   0   1   0   0   0   0   0   1   3   2
   0   1   0   0   1   0   0   0   0   1   3   5
   0   0   0   0   0   1   0   1   0   0   1   2
   0   0   0   0   0   1   0   0   1   0   2   6
   0   0   0   0   0   0   0   0   0   1   2   1
   0   0   0   0   0   0   0   0   1   0   2   1
   0   0   0   0   0   1   0   1   0   0   2   3
   1   0   0   0   0   1   0   0   0   0   2   3
   0   1   1   0   0   0   0   0   0   0   1   1
   0   0   1   0   0   1   0   0   0   0   3   1
   0   0   1   1   0   0   0   0   0   0   3   2
   0   1   0   0   0   1   0   0   0   0   1   2
   0   0   1   0   0   0   0   0   1   0   1   2
   0   0   0   1   0   0   0   1   0   0   3   3
   0   0   0   1   0   0   0   0   0   1   1   3
/*                          BOUNDS                                  */
  20  50  20  50  20  50  20  50  20  50  20  50

/*                      MESSAGE WEIGHTS                             */
   1   1   2   1   3   1   3   2   2   2
```

APPENDIX C

Model Representation Method

APPENDIX C

C.1 Method of Model Representation

The advantage of the Petri net representation is that the net can be dynamically modified during processing, i.e., modelling of hierarchical processes, interprocessor communications, models with different attributes and events within the same graph, and delays within the system [C.1]. This was the primary reason for choosing this representation for the simulation hierarchical control system.

Petri nets can be produced by a preprocessor and then used for simulating the system. The net is a virtual machine and can be coordinated with other virtual machines. The Petri nets have been used to model the following operating systems: LOGOS, 360 OS/MFT, and CDC 6400 Scope 3.2 [C.3.2]. The advantage of this representation is that it aids in determining what functions in the system can be distributed by using the top-down approach. It also assists in the evaluation process. These advantages were improvements over other techniques such as ISP, VDL, and other process structuring languages.

C.1.1 Petri Nets [C.3.1] and [C.3.3]

There are two types of nodes in the net, i.e., event and

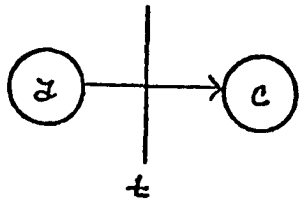
transition nodes. Events (e.g.,programs in execution) can accept tokens which implies the event is active, or has occurred. Transitions (or processes) show edges into the node (represented by a straight line) and edges out of the node. When a token (program or message event) occupies the event node, then the active transition may fire. The token is removed from the input node and placed upon the output node. More than one token can occupy an event node which could be used in later firings. What is not shown in the net is the effect of the flow through the transitions on the control variables and state tables. The process of occupying an event node may set the state variables, but the transition node can change the state variables as well as offer time delay through the network.

The Petri net primitives, shown in Figure C.1, are the T-transition, the F-transition and the J-transition. In the T-transition, the transition node t fires and after some time delay, the token moves from b to c. For example, let node b represent the arrival of a program in the system. After the execution of the program and after an appropriate time delay, the program is moved through the transition node to the nest node, (i.e., c in Figure C.1a).

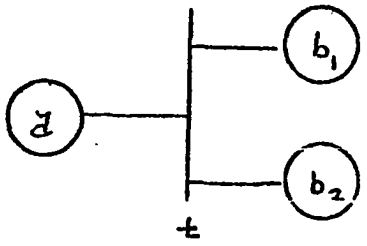
The F-transition, also called Fork, will move two tokens into the output nodes, b_1 and b_2 in Figure C.1b. The attributes of the two output nodes do not have to be the same. For example, two separate events must be initialized at the start of a time frame. One event can be the creation of the event

Figure C.1 Petri Network Primitives

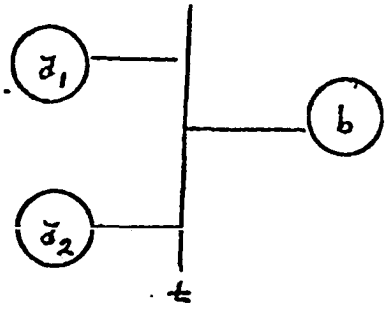
a. T-transition



b. F-transition (Fork)



c. J-transition (Join)



notice for next time and the other can be the initializing the execution of a processing element.

The J-transition, or Join, moves only a single token from two input nodes, a_1 and a_2 in Figure C.1c, to the output node b. For example, two jobs arrive in the system before the cpu is available, then the highest priority job will be selected when the cpu is ready. Since Petri nets do not provide a quantitative framework for expressing time of action, extensions to the representation are considered in the next section.

C.1.2 Macro E-Nets

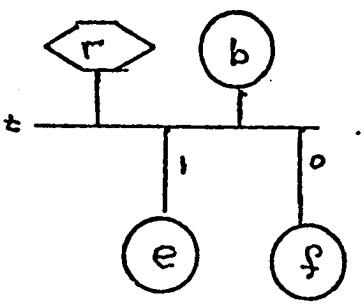
E-nets allow decision attributes, such as resolution procedures, to be incorporated into the flow to allow for time delays and the movement of more than one token through the transition. Macro nets reduce the network complexity [5.3.4].

Properties of the resolution procedures: reference and alter values of the attributes of the tokens as they flow through the net; reference, but not altering environment variables (global control variables and parameters); can refer to other events not associated with its transition; does not accept tokens.

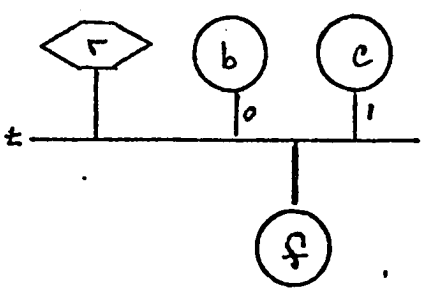
Two E-net extended primitives are the X-switch and the Y-switch, shown in Figure C.2. The X-switch, Figure C.2a, allows a resolution procedure, r, to determine the setting of the switch and can send the message from b to e or f. The location, r, has a third state referred to as "undefined." Neither output path is connected until after the resolution procedure alters the value of the input location from "undefined"

Figure C.2 E-Net Extended Primitives

a. X-switch



b. Y-merge



to zero or one. After firing the transition, the paths return to the "undefined" state. For example, an interrupt (token) a enters node b, the start of a new time frame. The resolution procedure tests the state vector to determine whether or not reconfiguration is necessary. If the system is to be reconfigured, then the token enters output node f after the resolution procedure has set the value to one. Otherwise, the token is moved to node e with the output value set at zero.

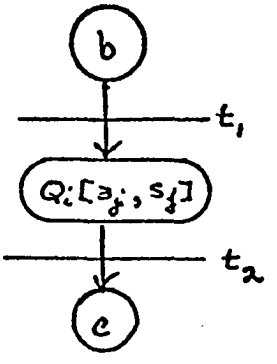
The Y-merge uses the resolution procedure, r, to determine which event, b or c in Figure C.2b, is to have priority; it is also used to prevent deadlocks. For example, at the start of a time frame in which reconfiguration is to take place, the events to recompute the processing element assignments and schedules must be executed before the programs in the processing element are selected for execution.

"Macro" nodes have been defined to represent processes such as AND, QUEUE, RESOURCE HANDLER, and CRITICAL SECTION nodes. The two Macro nodes, QUEUE and RESOURCE HANDLER, are defined for this study. The QUEUE macro in Figure C.3a is used to denote the process of producing a schedule for each processing element i. The "call" for the scheduler produces a request in node b. The node Q is a function of the state tables s_j, the run-time priorities, and the assignments a_j to the processing element. When the schedule has been completed, then a token moves into c.

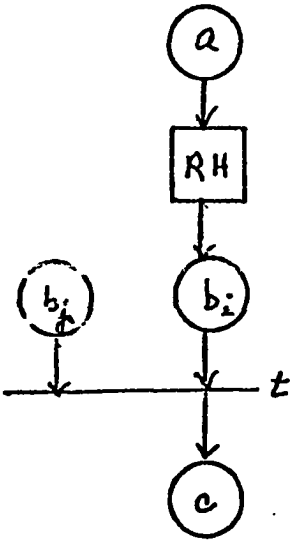
The RESOURCE HANDLER, in Figure C.3b will serve as the re-

Figure C.3 Macro Node Primitives

a. Scheduler



b. Resource Handler



configuration of the system for this study. The node b_j services as the "appropriate facility" to be used. Resources, the processing elements, are returned to node a as they become available and are reserved by node b_j through node b_i as they start execution.

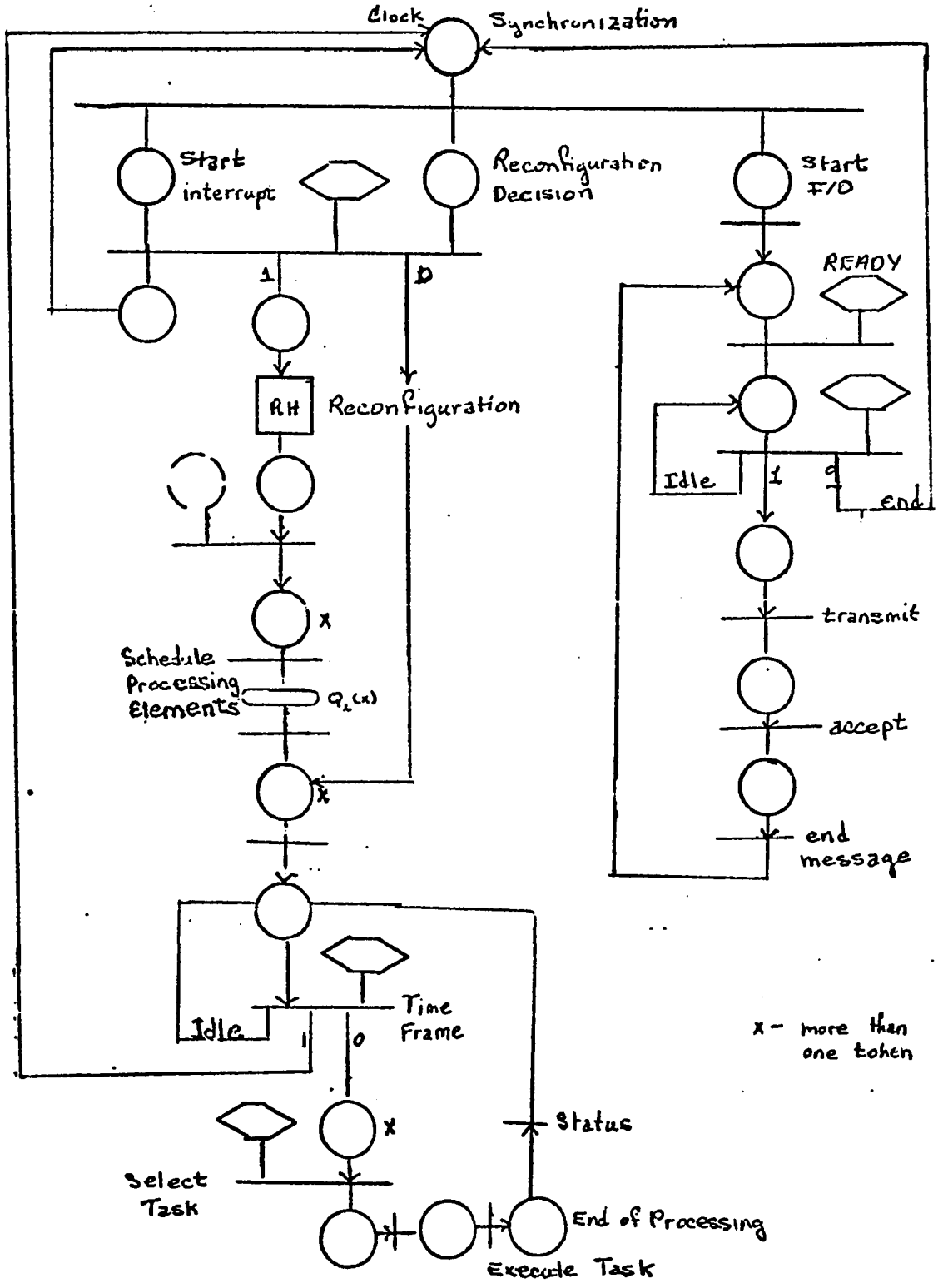
C.1.3 Information and Control Flow Model

The macro E-net in Figure C.4 combines the control and information flow of the process synchronization, bus traffic, arbitration method, reconfiguration and scheduling. In order to show the effects on the state variables, semantics can be added and used in the simulation. The resolution procedures are functions of the number of processing elements that are active and the status of reconfiguration.

C.1.4 Summary

Semantics can be incorporated in such a manner as to define the change in the state tables and control variables in the model. A "weak" correctness of the model has been attempted to prove that the model is pair event-wise deadlock free [5.3.5]. In this system, the deadlock issue is resolved due to the structuring of the message traffic and the writing over messages rather than reserving the bus. The primary advantage to be gained in this model is to combine the flow for different types of events, i.e., information and control flow through the hierarchical system.

Figure C.4 Macro E-Net of the Two-Level Controller for the Distribution Network



C.3 References

Petri and Macro E-nets:

1. Baer, J.L., "A Survey of Some Theoretical Aspects of Multiprocessing." ACM Computing Surveys, vol. 5, no. 1, Mar. 1973, pp. 30-80.
2. Baer, J.L., "Models for the Design, Sim, and Performance of Distributed-Function Architecture." Computer, vol. 7, no. 3, Mar. 1974, pp. 25-30.
3. Miller, R.E., "A Comparison of Some Theoretical Models of Parallel Computation." IEEE Trans. on Computing., vol. C-22, no. 8, Aug. 1973.
4. Noe, J.D. and Nutt, G.J., "Macro E-Nets for Representation of Parallel Systems." IEEE Trans. on Computing, vol. C-22, no. 8, Aug. 1973, pp. 718-727.
5. Agerwala, T. and Flynn, M., "Comments on Capabilities, Limitations and 'Correctness' of Petri Nets." Proc. on Conf. on Computer Architecture, 1973, pp. 81-86.