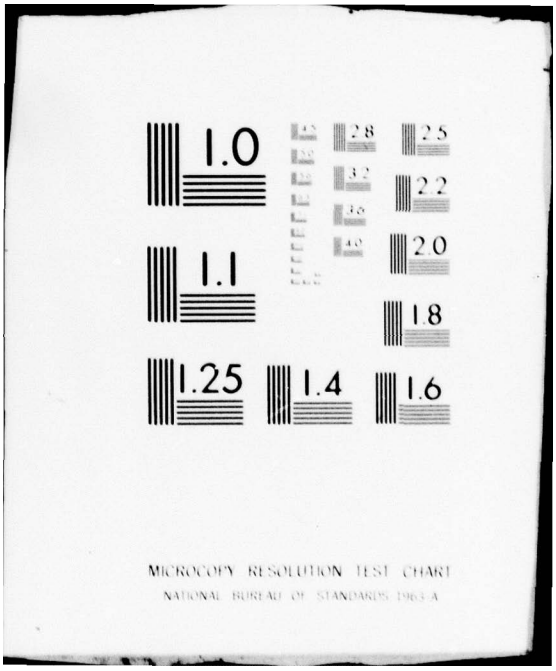
AD-A080 521 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
OPTIMAL MULTIPROCESSOR SCHEDULING OF PERIODIC TASKS IN A REAL-T--ETC(U)
DEC 79 W D SEWARD
UNCLASSIFIED AFIT/DS/EE/79-2 NL

1 OF 4
AD A080521


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

AD A080521

DDC FILE COPY

DDC
RECEIVED
FEB 11 1980
A

UNITED STATES AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

80 2 5 244

AFIT/DS/EE/79-2

OPTIMAL MULTIPROCESSOR SCHEDULING OF PERIODIC TASKS IN A REAL-TIME ENVIRONMENT

DISSERTATION

AFIT/DS/EE/79-2 Walter D. Seward

Capt USAF

DDC
RECEIVED
FEB 11 1980
A

Approved for public release; distribution unlimited

(14) AFIT/DS/EE/79-2

(6) OPTIMAL MULTIPROCESSOR SCHEDULING OF PERIODIC TASKS IN A REAL-TIME ENVIRONMENT,

(9) Doctoral thesis,

DISSERTATION

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

in Partial Fulfillment of the

Requirements for the Degree of (11) Dec 79

Doctor of Philosophy

(16) 2003 by (17) 03

(10) Walter D. Seward, B.S.E.E., M.S.E.E.

Capt USAF

(12) 308

Approved for public release; distribution unlimited.

Accession For
NTIS GRA&I
DDC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or special
A

022 225

AFIT/DS/EE/79-2

OPTIMAL MULTIPROCESSOR SCHEDULING OF PERIODIC TASKS IN A REAL-TIME ENVIRONMENT

by

Walter D. Seward, B.S.E.E.,M.S.E.E.

Capt USAF

Approved:

Gary B Lamont 4 Dec 1979
Chairman

Charles W Roark 4 Dec 1979

Alan A. Ross 4 Dec 1979

Thomas C. Hartrum 4 Dec 1979

Accepted:

J S Przemieniecki 4 Dec. 1979
Dean, School of Engineering

PREFACE

The research reported herein was performed while I was assigned to the System Avionics Division of the Air Force Avionics Laboratory at Wright-Patterson Air Force Base, Ohio.

I wish to express my deep graditude to my advisor, Professor Gary Lamont, for his invaluable guidance and support throughout my research work. I also gratefully acknowledge the valuable comments and suggestions of the other members of my committee, Professors George Orr, Charles Roark, Alan Ross and Thomas Hartrum.

Finally, I am especially indebted to my wife, Sadie, and my sons, Tony, Steve, and Dan for their sacrifices and understanding throughout, what has turned out to be, a long course of studies and research.

TABLE OF CONTENTS

		Page
PREFACE		ii
LIST OF FIGURES		v
LIST OF TABLES		viii
ABSTRACT		ix
1.	INTRODUCTION TO SCHEDULING PERIODIC TASKS	1
	1.1 Introduction	1
	1.2 The General Scheduling Problem	6
	1.3 Survey of Periodic Job Scheduling	13
	1.4 Problem Statement and Approach	23
2.	CLASSIFICATION OF PERIODIC TASKS	30
	2.1 Introduction	30
	2.2 Schedules of Periodic Tasks	31
	2.3 Restricted Regions for Periodic Tasks	39
	2.4 Restricted Regions Revisited	58
	2.5 Compatibility Classes of Periodic Tasks	68
	2.6 Load Consistent Maximal Compatibles (LCMC)	89
	2.7 Summary	90
3.	AN OPTIMAL ALGORITHM FOR UNIPROCESSOR SCHEDULING OF INDEPENDENT PERIODIC TASKS	92
	3.1 Introduction	92
	3.2 Previous Results for Single Processor Scheduling	93
	3.3 Accessible Regions for Periodic Tasks	96
	3.4 Valid Schedules for Periodic Tasks	103
	3.5 Critical Interval for Periodic Task Scheduling	106
	3.6 Families of Valid Schedules	116
	3.7 Accessible Regions as Linear Inequalities	120
	3.8 Mixed Integer Programming Solution to Scheduling	127
	3.9 Vectors of Schedule Elements	137
	3.9.1 Sequences of Schedule Elements	140

3.9.2 Schedule Vectors with the LCC Reference Job 157
3.10 Inconsistent Schedule Vectors 168
3.11 Equivalence Classes of Schedules 177
3.12 Consistent Enumeration Trees 185
3.13 An Optimal Scheduling Algorithm 194
3.14 Optimal Schedules 201
3.15 Summary 202

4. MINIMAL MULTIPROCESSOR SCHEDULES FOR INDEPENDENT PERIODIC TASKS 203

4.1 Introduction 203
4.2 Minimal Coverings of LCMC 205
4.3 Essential Compatibles and Reduced Job Sets. 216
4.4 Set Systems of LCMC 233
4.5 Addmissible Partitions 241
4.6 Minimal Processor Scheduling Algorithm ... 252
4.7 Upper and Lower Bounds for Minimal Processor Schedules 259
4.8 Summary 261

5. SCHEDULING NON-INDEPENDENT PERIODIC TASKS 262

5.1 Introduction 262
5.2 Task Systems of Non-Independent Jobs 262
5.3 Minimal Processor Schedules of Non-Independent Job Sets 264
5.4 Summary 274

6. CONCLUSIONS AND RECOMMENDATIONS 275

6.1 Introduction 275
6.2 Summary and Conclusions 275
6.3 Reccommendations and Possible Extensions .. 277
6.3.1 Implementation and Analysis of Algorithms 277
6.3.2 Generalization of the Problem Characterization 279
6.4 Final Remarks 281

BIBLIOGRAPHY 282

APPENDIX A: AN INDEX OF DEFINITIONS AND TERMINOLOGY ... 288

APPENDIX B: GROUP THEORETIC NOTATION, DEFINITIONS, AND THEOREMS 291

LIST OF FIGURES

Figure		page
1-1	Advanced Navigational System	4
1-2	Example Timing Diagram for a Periodic Job	14
1-3	Allowable Assignment of j_2	20
1-4	Unallowable Assignment of j_2	20
1-5	A Preemptive Schedule for j_1	26
2-1	Restricted Regions That Contain Active Intervals	42
2-2	The n+1st Reference Frame of Job j_k	44
2-3	A Three Job Schedule and One Subschedule	57
2-4	The Relationship of Restricted Regions $r_{ik}(m)$ to $r_{ik}(n)$	59
2-5	Restricted Regions That Contain Active Intervals of j_1	59
2-6	Restricted Regions	61
2-7	All Restricted Regions of j_2 Relative to Job j_1	62
2-8	Restricted Regions of $\{j_2, j_1(t_{01})\}$	62
2-9	Restricted Regions for j_k Relative to j_i	64
2-10	Binary Tree Representation of $f(x_1, x_2)$	76
2-11	Tree Structure for Incompatibility Function	87
3-1	Accessible Regions of j_2 Relative to j_1	99

3-2 The Critical Interval of j_i and j_k 107

3-3 Valid Schedule of "Pseudo" Job $j'_{1,2}$ and Job j_2 110

3-4 The First Period of a Valid Three-Job Schedule 117

3-5 Schedule Period for $\{j_1, j_2 (t_{02} - t_{01}), j_3 (t_{03} - t_{01})\}$ 118

3-6 Schedule Period for $\{j_1 (t_{01} + T_1 - t_a), j_2 (t_{02} + T_2 - t_a), j_3 (t_{03} - t_a)\}$ 119

3-7 Schedule Period for Shifted Jobs j_2 and j_3 119

3-8 Enumeration Tree for a Three-Job Set 131

3-9 Job Pair Schedule 141

3-10 Job j_1 Repeats as Reference Job 150

3-11 One Active Interval of j_k within the Idle Interval of j_i .. 151

3-12 One Active Interval of j_i Between Active Intervals of j_k .. 153

3-13 Relation of Active Intervals of j_i to j_1 156

3-14 Sequence of j_2 and j_3 within Three-Job Schedule Relative to j_1 158

3-15 Three Job Valid Schedule 179

3-16 An Optimal Scheduling Algorithm 200

4-1 Phase I - Minimal Processor Scheduling Algorithm 226

4-2 Semilattice of Set Systems of LCMC's 240

4-3 Minimal Processor Scheduling Algorithm (Phase II) 258

5-1 A Precedence Graph of Job Set K 263

5-2 Precedence Graph of Example Job Set 269

5-3 Precedence Graph of Example Job Set 270

LIST OF TABLES

Table		page
3-1	Start Time Differences for $\{j_1, j_2, j_3, j_4\}$	124
3-2	Start Time Differences $t_{02} - t_{01}$ vs. Schedule Element s_{12}	144
5-1	Schedule Elements of Example Job Set	272

ABSTRACT

This investigation has addressed the problem of constructing a nonpreemptive schedule that requires the minimal number of processors for a given set of periodic tasks. Each periodic task is characterized by an integer period and an execution time. It is assumed that the period between each initiation and termination of a task must not vary once the first is specified.

A compatibility relation is defined on the set of tasks such that any pair of tasks may be scheduled on the same processor only if they are compatible. This compatibility relation is based on the greatest common divisor of the task periods and the sum of the task execution times. The tasks are classified into maximal compatibles whereby no two tasks in any maximal compatible set exclude each other from a schedule on the same processor. An equivalence class of schedules of a given subset of tasks is defined and an optimal algorithm for constructing a single processor schedule for a given subset of a maximal compatible is developed. This algorithm is optimal in the sense that it will construct a single processor schedule if one exists for any subset of a maximal compatible.

A lower bound on the number of processors required for the given task set is defined based on the minimal covering of maximal compatibles of the task set. A lattice structure is then developed from all of the irredundant coverings of maximal compatibles. The lattice structure contains every possible combination of tasks which may have a valid

schedule. An algorithm is defined which constructs a minimal processor schedule for a given set of tasks. This algorithm is based on the optimal single processor algorithm and the implicit enumeration of the possible schedules which do not violate the periodicity constraints of the task set. The algorithm determines both upper and lower bounds on the number of processors required. These bounds converge to a common value as the algorithm converges on an optimal schedule, thereby allowing the system designer the alternative of choosing a suboptimal schedule which is within known worst-case bounds of an optimal assignment.

CHAPTER 1

INTRODUCTION TO SCHEDULING PERIODIC TASKS

1.1 Introduction

Recent advances in computer technology, particularly in the area of increased cost effectiveness of minicomputers and microprocessors, has intensified interest in distributed systems for real-time control applications (often refered to as process control). For our purposes, a distributed system is any collection of resources (sensors, processors actuators, etc.) working asynchronously but cooperatively toward a common goal. The advantages of a distributed system for real-time applications include increased system performance, availability, and flexibility (Bib,Jen,Ser).

Although system performance is not a linear function of the number of processors in a system, due to contention among the hardware and software resources of the system, distributed systems usually give a significant reduction in the incremental cost to performance ratio. Furthermore, in some environments, the constraints of volume, weight, and power requirements frequently preclude the use of a larger single processor to meet the performance objectives of the distributed system. Finally, distributed systems offer the capability to implement more powerful control strategies due to more efficient allocation of tasks to the separate processors in the system.

The availability of single processor systems is dependent on the

individual elements of that system. Distributed systems of multiple resources of a given type can provide greater reliablilty at a given cost than can a system with a single processor. Failure of resources in a distributed system may allow continued operation of the system in a degraded, but acceptable, mode.

The inherent modularity of distributed systems can enhance the flexibility of the system to changes or addition of different sensors, or a change in processor's capabilities. The ease with which distributed systems can be expanded or the system configuration changed makes distributed system designs more adaptable to advances in component technology or changes in system requirements.

Examples of real-time control applications which could benefit from such distributed architectures include avionics, space, and ship board command and control systems, and systems for the control and monitoring of industrial processes (Gonl,Bib,Jos).

As an example of such a system, consider the problem of controlling the quantity of chemicals to be mixed as well as the temperature and pressure inside of a reaction tank on the basis of various measurements (Liu). Further, each of the adjustments require different frequencies. That is, the adjustments in the quantity of chemicals must be carried out more frequently, while the temperature and pressure require less frequent adjustment.

The problem is to schedule the usage of the computer resources so that <u>all</u> of the requests by each of the processes (adjustments of the quantity of chemicals, the temperature, and the pressure) are satisfied.

For a more detailed example, consider a simple representation of the navigational subsystem of an aircraft's avionic system (US,BF,Gel).

Such a navigational system is designed to provide redundant navigational data from a group of different sensors - all of which are subject to random errors caused by environmental conditions, equipment condition, and system design. In the fully operational mode, the system uses a Kalman Filter to determine the navigational information from all of the available input data with minimum error and to improve the system accuracy by estimating the system error states and generating compensation signals. The system, in addition, has the capability of functioning in a degraded mode with one or more of the subsystems not operational.

In addition to the navigational elements, the system includes a Built In Test (BIT) system to monitor the sensors and to provide for automatic upmoding and downmoding of the system based on the functioning of the individual system elements.

Figure 1-1 contains a functional diagram of such a navigational system (US).

We will assume that the navigation system consists of an Inertial Measurement Unit (IMU), Doppler radar to measure the ground speed of the aircraft, a radio position-fixing system such as LORAN, and an Air Data system to calculate barometric altitude, true air speed, and free air temperature. The IMU is used to measure the velocity increments of the aircraft along the axis of the IMU platform and to measure the orientation angles which are used to determine the Doppler and Air Data orientations. The radio position-fixing system provides position information with respect to known positions of transmitter stations.

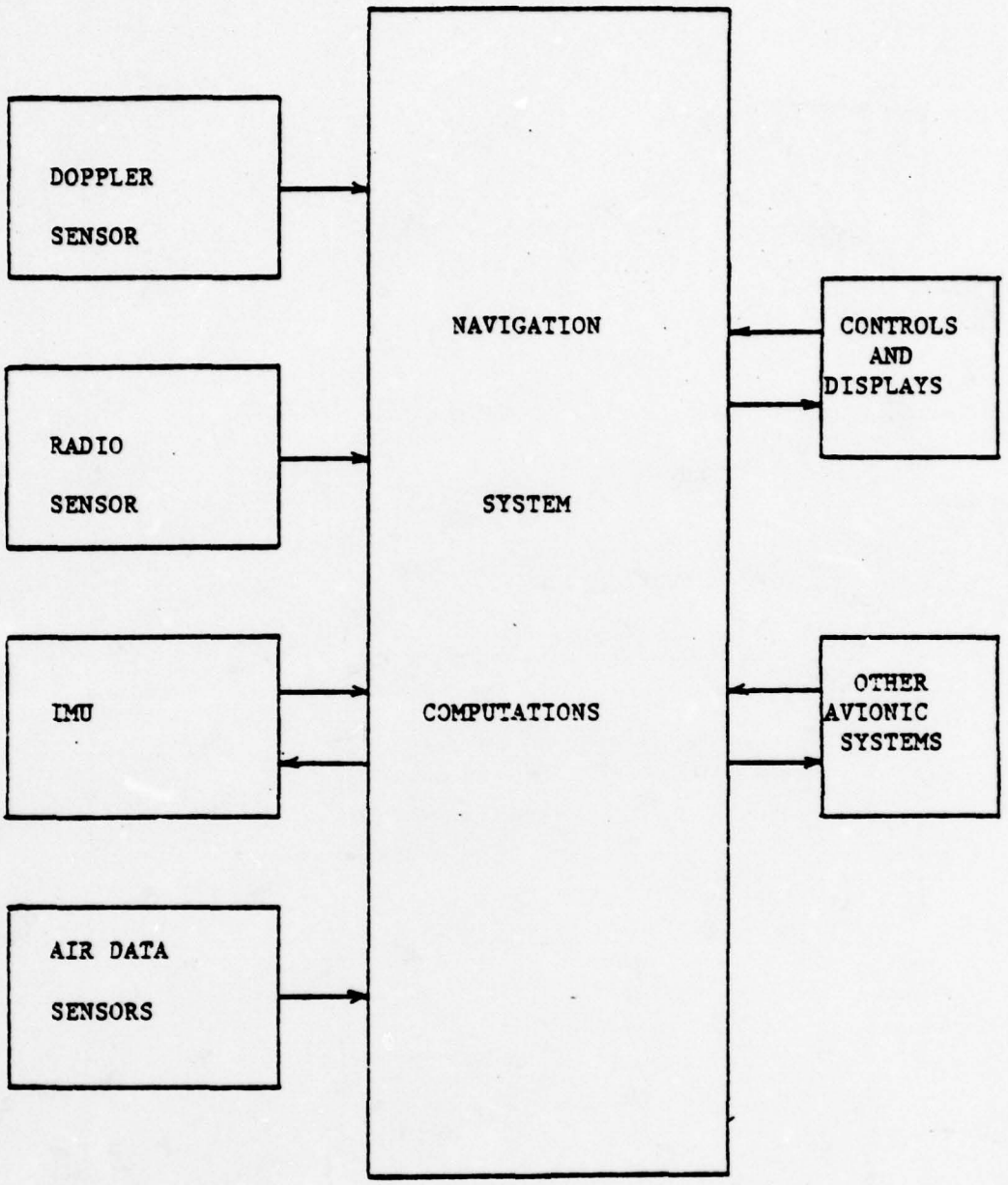


Figure 1-1 Advanced Navigational System

The output signals of these navigation sensors are not directly useful as navigation information, and the signals must be incorporated into navigation equations to calculate the state of the aircraft at any given time. Each of these calculations must be performed on a periodic basis, and each computation requires a known maximum execution time.

The results of the calculations of each of the sensors is then incorporated through a measurement matrix of a Kalman filter for the final determination of the state of the aircraft. Some of the results of the navigational calculations are also used by other aircraft systems such as the Automatic Flight Control System (AFCS) and the aircraft's Stability Augmentation System (SAS) - these systems are also periodic in nature.

While the exact iteration rate and computation time of each of the navigational functions to be performed depends on the particular system, the following table presents a representative list of function repetition rates and execution times: (US,BF)

Function	Frequency	Time (msec)
Air Data System		
Barometric Altitude	15	6
True Airspeed	10	6
Free Air Temperature	3	6
IMU	20	4
LORAN	20	5
Doppler Radar	10	1
Kalman Filter	2	25
BIT	1	5

As with the previous example, our problem is to determine how all of the tasks, which have apparently unrelated frequencies and execution times, listed above can be accomplished in such a way that the required system functions are satisfied (i.e. the periodic request rates

and the execution time requirements of each task) with the resources available.

As each of the above examples illustrates, the system in question is required to respond to external stimulus such as sensor inputs and perform the necessary calculations at a sufficient speed to respond to the stimulus and maintain the environment within some set of desired limits. Real-time applications such as these are generally characterized by collections of periodic tasks for which the periods, execution times, and precedence constraints are known a-priori (Aro). It is this problem which we will investigate.

Before we begin, a brief overview of this chapter is provided. We will briefly present and discuss some of the terminology and definitions, both those that are used in the discussion of the general problem and those that also apply to the specific problem of scheduling periodic tasks. For the benefit of the reader, definitions of various terms are indexed in Appendix A, "An Index of Definitions and Terminology."

In Section 1.3, a summary of previous results in the area of scheduling periodic tasks on multiprocessor computing systems is given. This will be followed by an informal detailed statement of the problem and a discussion of the rationale for some of the assumptions.

1.2 The General Scheduling Problem

In this section, the terminology and definitions of a scheduling model is presented in the most general terms. The limitations that are applicable to the less general problem of scheduling tasks on computing systems will be discussed, and the constraints which apply equally well

to the problem of scheduling periodic jobs presented. We will then briefly discuss the complexity of the general scheduling problem and present a summary of some of the results for specific problems. The specific terminology applicable to the problem of scheduling periodic jobs and the results that have been previously obtained will be presented.

In most general terms, the scheduling problem with which we will be concerned is the allocation of available resources, over a period of time, to perform a given set of jobs subject to a known set of constraints. The resources will be allocated in such a way as to optimize or tend to optimize a given performance measure. The performance measure, some examples of which are discussed in the following paragraphs, for each problem is determined by the goals of the system development.

A general model for defining such a problem is described by considering the resources, a task system, the sequencing constraints,and the performance measure. Our discussion of the structure of a general model is based on a presentation of a general model by Coffman (Cof).

For the most general problems, the resources of the system are contained in a collection of resource types, $\{R_i\}$. Each of the sets of the different types of resources, R_i, contains m_i elements. Such resource types might include memory, both primary and secondary, input/output devices including sensors for sampling the environment external to the system, or any other hardware or software resource required in the operation of the system.

For the problems with which we will be concerned, the resources

will consist of a set of processors, $P=\{P_1, P_2, \ldots, P_m\}$. The processors in the set may be identical, identical in functional capability but different in speed, or different in both functional capability and speed. We will assume in this investigation that the set of processors are identical, or as this condition is frequently refered to, a homogeneous set of processors. A homogeneous set of processors is assumed for this investigation because it provides a basis upon which a theoretical development can be built prior to extending the results to a more general case.

In the most general sense, the task system, S, for a given set of resources is defined as the tuple $(K, \prec, [E_{ij}], \{R_j\}, \{w_i\})$ where the elements of the task system are defined as follows:

1. $K=\{j_1, j_2, \ldots, j_n\}$ is a set of jobs or equivalently tasks to be executed.

2. Defined over the set K is a partial order, $\prec$, which specifies the precedence constraints among the tasks. That is, $j_i \prec j_k$ signifies that the execution of j_i must be completed before j_k may begin. A partial order is defined to be a set of ordered pairs $\{(j_i, j_k), (j_m, j_p), \ldots\}$ such that if $j_i \prec j_k$ then (j_i, j_k) is in the set of ordered pairs. The partial order for a given job set may be empty, in which case the jobs in the set are said to be independent.

3. The matrix $[E_{ik}]$ is an n by m matrix of execution times of the job j_i on the processor P_k. In the case where j_i cannot be executed

on processor P_k then we represent this fact by making E_{ik} infinite. But, for our discussion, we will assume that each of the processors available is capable of executing each of the jobs in the job set. Furthermore, since we have assumed a homogeneous set of processors, we can represent the execution times of each of the jobs independent of the processor to which it may be assigned, i.e., the execution time for j_i is defined by E_i.

4. The collection of resource sets, $\{R_i\}$, defines for each task the amount of resource of type R_i required by job j_k. In the problems that we will consider, the only resource with which we will be concerned is the allocation of a processor, P_i, to a job, j_k, each time j_k is required to execute. The resource set for our problem will consist of a set of processors, $\{P_j\}$.

5. The weights $\{w_i\}$, one weight corresponding to each of the jobs in the job set, are arbitrary weighting functions of the schedule properties associated with the job j_i. For example, there may be associated with each job a penalty for assignment prior to a given time, t_0, and a penalty for any assignment after a given time, t_1, in a given schedule. There is a cost weighting for that job which is a function of the job's scheduled time and the weights associated with any start time not within the bounds of t_0 and t_1. We will assume that there is no set

of weights explicitly associated with the elements of the job set for the problems of interest to us. Although, as we will show in a later chapter, there is an implicit weight for any job start time not within zero and the value of the job's period minus its execution time. In fact, the weighting associated with this constraint precludes a valid schedule even existing for such a job start time.

In light of the restrictions to the general model that we have discussed above, we can represent the class of problems in which we are interested by the following task system $S = (K, \prec, [E_i], \{P_j\})$.

Given the resources and the task system, there is associated with the scheduling algorithms certain constraints relative to the assignment of the jobs to the processors.

First of all, when a task cannot be interrupted once it is initiated, i.e., it is allowed to run to completion, then the schedule is said to be <u>nonpreemptive</u>. This is in contrast to the a <u>preemptive</u> algorithm in which the execution of a task may be interupted prior to its completion if a job of higher priority requests execution. The preempted task then resumes execution at the point at which it was suspended once it becomes the highest priority task requesting execution.

There are other constraints that may pertain to a given scheduling strategy. If all of the characteristics of the scheduling environment are available a-priori, then the environment is termed <u>deterministic</u>. A <u>stochastic</u> scheduling environment on the other hand is based on the probabalistic characteristics of the jobs. That is the exact job charateristics are not known; only the statistics of the jobs' requirements for execution.

A schedule for a given set of jobs is said to be feasible or valid if all of the constraints of the task system are satisfied. Thus, for a schedule to be valid, no conflicts may occur within any of the job assignments. A conflict occurs when two or more jobs are scheduled to be executing on the same processor at the same time. In addition, for a schedule to be valid, all of the precedence constraints relative to the interdependence of the jobs must be satisfied.

The final element in the general model defined above is the performance measure. There are several performance measures in common usage for scheduling jobs on computing systems. Among the performance measures, maximum finishing time and mean weighted finishing time for a schedule are the most frequently used (Cof). Other performance measures that are often used include minimization of weighted tardiness and minimization of weighted lateness both of which are based on an established due date for each job. The dual problem of schedule length minimization for a given set of k processors is that of finding the minimum number of processors required for a given schedule length (Cof,CD).

Throughout this discussion we have referenced an algorithm for scheduling the set of jobs. We would of course prefer that the algorithm be efficient. An algorithm is said to be efficient if the number of elementary steps is bounded by a polynomial function of the problem size - in this case the number of jobs to be scheduled (Cof). There is, unfortunately, a significant collection of previous results that indicate that the general scheduling problem belongs to a class of problems called NP-Complete (Sav,Kar).

Within the class of problems known as NP-Complete, it is known

that in terms of complexity, all of the problems in the class are equivalent in the following sense:

> If there exists a polynomial time solution to any one of the problems in the class, then there exists a polynomial time algorithm for all of the NP-Complete problems.

Among those problems in the class of NP-Complete problems are some that have been studied extensively and for which no polynomial time algorithm has yet been found. Prime examples are the Traveling Salesman Problem, The Knapsack Problem, and Satisfiability of Boolean Functions (Sav,Kar).

In the general area of scheduling, efficient algorithms have been found only for specially structured problems, such as equal execution time tasks with a tree structured precedence relation (Cof,CD).

For seemingly simple problems such as the nonpreemptive scheduling of independent tasks with arbitrary execution times on a fixed number of two or more processors, the minimization of the maximum finishing time is known to be an NP-Complete problem (Cof).

It seems doubtful therefore that an efficient algorithm will be easily found for our particular problem. We will address this point in greater detail in the last section of this chapter. (The interested reader can find additional information on NP-Complete problems in (BGJ,CD,Cof,GJ1,GJ2,GJ3,Kar,Sha,Ull)).

With this general formulation of the scheduling problem and

terminology defined above, we are now prepared to discuss the problem of scheduling periodic jobs on the minimal number of processors.

1.3 Survey of Periodic Job Scheduling

There has been a significant amount of research done on the topic of multiprocessor scheduling in general, but we will postpone a discussion of general techniques and results until Chapter 4 and limit our discussion of multiprocessor scheduling in this chapter to results that apply specifically to periodic or time-critical tasks. A time-critical task or time-critical process is a periodic task of known frequency and execution time such that each activation of a given task must complete prior to a known deadline for that task. If, in addition, it is assumed that failure of the task to complete prior to its deadline results in catastrophic failure of the system, it is refered to as a hard-real-time environment. (When it is sufficient that failure to meet the deadline occurs in less than some average number of cases, the environment is said to be soft-real-time.) The deadline of a given task must not exceed the period of that task; and the period of a task is frequently chosen as the deadline. At this time we will concentrate on the previous work in the area of multiprocessor scheduling of periodic tasks.

We can represent the time histories of each of the jobs in the set as well as each of the processor schedules by means of a timing diagram called a Gantt chart (CD). A Gantt chart consists of a time axis for each processor with intervals marked off and labled with the name of the task being processed. Each execution of each task is represented by a rectangle whose length corresponds to its execution

time. The Gantt chart gives an informal and intuitive notion of a schedule and the operation of the periodic jobs over time. The figure below, for example, represents two periods of a periodic job.

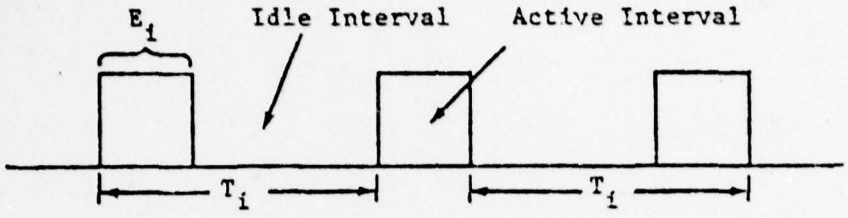


Figure 1-2 Example Timing Diagram for a Periodic Job

The figure above illustrates the two distinct phases of a periodic task. These phases are the active interval or equivalently the active period and the idle interval or idle period. It is during the active interval that a job is performing the requested information processing and at all other times it is idle. A request for the execution of a given job occurs once each period of that job.

There have been relatively few results published which address the problem of scheduling time-critical tasks. The bulk of the previous work has been concerned with defining schedules for time-critical tasks on a single processor. The work on single processor scheduling includes papers by Liu and Layland (LL), Serlin (Ser), and Labtoulle (Lab). We will discuss in detail the work of these authors in the area of multiprogramming of periodic tasks in Chapter 3. At this time we will discuss the previous work in the area of multiprocessor scheduling of periodic tasks.

The problem of multiprocessor scheduling of periodic jobs has been previously considered by Dhall and Liu (Dha,DL), by Dabholkar

(Dab), and by Gonzalez and Soh (GS,Soh). Each group of authors has taken a different approach to defining the problem and the resulting methodology and schedules.

Dhall and Liu examined the problem of determining the minimal number of identical processors required for a preemptive schedule of a given set of periodic time-critical tasks. The tasks in the set were assumed to be independent with perodic requests for execution of each task and a constant interval between the requests of a given task. It was further assumed that each request for task execution must be completed prior to the deadline for that request. The deadline for each request is defined to be the next request for the same task, i.e., the period of the task.

Two heuristic algorithms which guarantee that all of the deadlines would be met were examined and the worst-case behavior of these algorithms relative to the optimal number of processors required for a given job set were determined. Each of the algorithms defined were <u>static</u> or <u>fixed priority algorithms</u>, where a fixed priority algorithm is one in which the priority is assigned to each job and not to each request for a given job. Hence, for the set of jobs all of the relative priorities of the jobs are known before hand and they will not change during the execution of the system. Any request for execution of a job with a higher priority will have precedence over all requests for the execution of jobs with lower priority. Thus, in a preemptive scheduling algorithm, a request for a job with a higher priority than the job presently executing on a given processor will cause the lower priority job to be preempted and execution to begin on the higher priority job. The preempted job will resume its execution at the point

it was preempted when it once again is the job with the highest priority waiting to complete its request for execution.

For the two algorithms considered by Dhall and Liu, the priorities were assigned by non-decreasing rate of request. Thus the job with the highest priority is that job in the job set with the highest request rate while the job with the lowest priority is the job with the lowest request rate. Such an ordering is call a Rate Monotonic ordering.

The first algorithm considered is called the Rate-Monotonic-Next-Fit Scheduling (RMNFS) algorithm. According to this algorithm, the tasks are arranged in order of nonincreasing request rates. The tasks are assigned to the processors in this order. When each task is to be assigned, the feasiblity of the schedule of the last processor created, if the task in question were assigned, is determined by using the results of a theorem proven by Liu and Layland (LL). This theorem defines a necessary and sufficient condition for determining the feasibility of a multiprogramming schedule for a given set of periodic tasks. If the task to be scheduled does not result in an infeasiblity of the schedule considered, then it is assigned to that processor. If it does prevent a feasible schedule from existing on that processor, then another processor is created and the task assigned to the newly created processor. Each task is considered one at a time in the order of the established priority until all of the tasks have been scheduled. The order in which the existing processor schedules are examined severly restricts the possible feasible schedules that will be examined. The next algorithm they presented permits more flexible schedules.

This algorithm is called the Rate-Monotonic-First-Fit Scheduling

it was preempted when it once again is the job with the highest priority waiting to complete its request for execution.

For the two algorithms considered by Dhall and Liu, the priorities were assigned by non-decreasing rate of request. Thus the job with the highest priority is that job in the job set with the highest request rate while the job with the lowest priority is the job with the lowest request rate. Such an ordering is call a Rate Monotonic ordering.

The first algorithm considered is called the Rate-Monotonic-Next-Fit Scheduling (RMNFS) algorithm. According to this algorithm, the tasks are arranged in order of nonincreasing request rates. The tasks are assigned to the processors in this order. When each task is to be assigned, the feasiblity of the schedule of the last processor created, if the task in question were assigned, is determined by using the results of a theorem proven by Liu and Layland (LL). This theorem defines a necessary and sufficient condition for determining the feasibility of a multiprogramming schedule for a given set of periodic tasks. If the task to be scheduled does not result in an infeasiblity of the schedule considered, then it is assigned to that processor. If it does prevent a feasible schedule from existing on that processor, then another processor is created and the task assigned to the newly created processor. Each task is considered one at a time in the order of the established priority until all of the tasks have been scheduled. The order in which the existing processor schedules are examined severly restricts the possible feasible schedules that will be examined. The next algorithm they presented permits more flexible schedules.

This algorithm is called the Rate-Monotonic-First-Fit Scheduling

(RMFFS) algorithm. As with the RMNFS algorithm, the tasks are considered for scheduling in descending order of request rate, and each task is considered one at a time until all of the tasks in the job set have been assigned. But, for the RMFFS algorithm, all of the existing processors and the feasibility of their schedules, if the task in question were assigned to that processor, are considered in the order in which the processors were "created" until a processor is found that will not have the feasibility of its schedule changed with the assignment. A new processor is created only if no feasible schedule can be found on the existing processors.

The primary consequence of the work of Dhall and Liu is in the results of two theorems which define worst case bounds for the number of processors required for a feasible schedule for each of the two scheduling algorithms considered. The two theorems are stated below:

Theorem 1.1 (Dha,DL) Let N be the number of processors required to feasibly schedule a set of tasks by the RMNFS algorithm, and N_0 the minimum number of processors required to feasibly schedule the same set of tasks. Then as N_0 approaches infinity,

$$2.4 \leq N/N_0 \leq 2.67.$$

Theorem 1.2 (Dha,DL) Let N be the number of processors required to feasibly schedule a set of tasks by the RMFFS algorithm, and N_0 be the minimum number of processors required to feasibly schedule the same set of tasks. Then, as N_0 approaches infinity,

$$2.0 \leq N/N_0 \leq 4(2^{1/3})/(1+2^{1/3}) = 2.3.$$

It is conjectured that the upper bound in Theorem 1.2 can be reduced to equal 2 (Dha,DL). In either case, the bounds determined for the two algorithms considered indicate that for some sets of tasks at least twice as many processors as the minimum number may be required for a feasible schedule to exist if either of these algorithms are employed.

Both of the algorithms considered by Dhall and Liu are preemptive. Other published results in multprocessor scheduling of periodic tasks have been in the area of nonpreemptive scheduling.

Dabholkar (Dab) investigated the problem of nonpreemptive scheduling of independent periodic jobs on a fixed number of processors. The jobs were assumed to have integer periods and the initial request for initiation of each job was at $t = 0$. The deadline for each job was defined to equal its period.

First, Dabholkar analyzed the complexity of scheduling periodic jobs on a fixed number of identical processors. He considered both the list scheduling and dynamic scheduling problems and demonstrated that the determination of a schedule in which every request for computation is satisfied before its deadline belongs to the class of NP-Hard problems. The class of NP-Hard and NP-Complete problems are related by the notion of reducibility (SH).

Given problems L_1 and L_2, L_1 <u>reduces</u> to L_2 if and only if there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time (SH).

A problem L is NP-Hard if and only if satisfiability of a boolean function reduces to L. A problem L is NP-Complete if and only

if it is NP-Hard and L is an element of the class NP. Hence, there are NP-Hard problems that are not NP-Complete.

Dabholkar also analyzed nonpreemptive scheduling algorithms and determined that when a job set has integer frequency distribution and equal execution times, the rate monotonic priority list scheduling algorithm is optimal. That is, it will determine a feasible schedule for the jobs and the given processors if such a schedule exists. In addition, he examined dynamic nonpreemptive scheduling algorithms and defined an algorithm which is optimal for single processor scheduling of jobs with equal execution times. As pointed out previously, the deadline for each job was defined to equal its period.

Soh and Gonzalez (GS,Soh) addressed the problem of determining a nonpreemptive, minimal processor schedule for a given finite set of independent periodic tasks with know integer request periods and known execution times for each request. The schedules were determined based on the additional assumptions that the processors were identical and the deadline associated with each job's request for execution was equal to the execution time for that job. That is, each request of each job must be honored immediately or the deadline for that request will be exceeded. It was also assumed that the computational requirements for each job in the set can be satisfied by a single processor without violating the periodicity of the jobs' requests.

Soh defined an algorithm for determining a minimal multiprocessor schedule for a given set of periodic jobs with a binary frequency distribution. That is, the set of tasks when ordered by nondecreasing frequency have a frequency distribution defined by the recursive equation $f_i = 2f_{i+1}$.

As an example, consider a three job set

Job	Frequency
j_1	8
j_2	4
j_3	2

which has a binary frequency distribution ($f_2 = 2f_3$, $f_1 = 2f_2$).

The scheduling of the job sets so defined was further constrained by a requirement that the execution time of the first active interval of each job (after the first) which is assigned to a given processor must be initiated upon the completion of some active interval of a job previously assigned to that processor. For example, if there is a processor with a job j_1 previously assigned, then a job j_2 may be assigned as shown in figure 1-3.

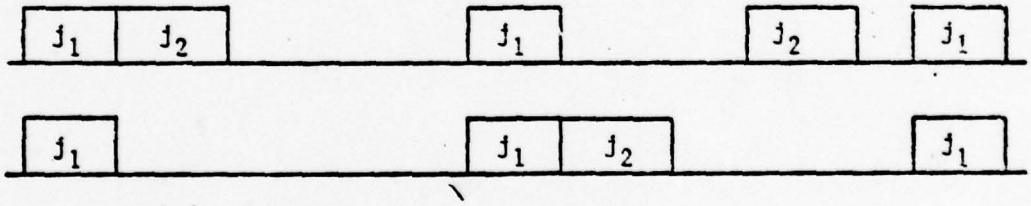


Figure 1-3 Allowable Assignments of j_2

but not assigned as illustrated below.

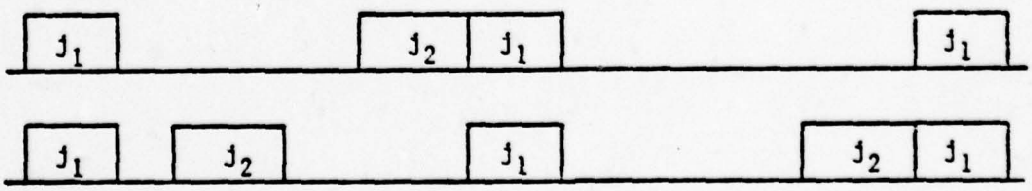


Figure 1-4 Unallowable Assignments of j_2

Soh defined a Frequency-Decreasing Priority (FDP) assignment as an assignment order in which the jobs are assigned to a processor in frequency decreasing order. That is, the job with the higher frequency is given the higher priority in the assignment process. This static priority is the same as that used by Dhall, but for the nonpreemptive scheduling environment. It determines only the order in which the jobs will be scheduled and not the priority of the tasks during execution on a processor.

For the job sets as defined above, Soh was able to show that an FDP assignment is optimum in the sense that no other static assignment rule can schedule a given job set which cannot be scheduled by the FDP assignment. This relationship of the FDP assignment rule to other assignment rules is stated in the following theorem:

Theorem 1.3 (Soh,GS) If a feasible schedule exists on a processor for a job set which has a binary frequency distribution, $f_i = 2f_{i+1}$, then the FDP assignment is also feasible on a processor for that job set.

Based on this theorem, Soh defined an optimal algorithm for scheduling a given set of jobs, as defined, on a minimal number of processors given the constraints on the job set and the allowable assignments defined previously.

The algorithm defined by Soh is analagous to the RMFFS algorithm of Dhall. Each task in the job set is considered for assignment to a processor in descending order of frequency. Each existing processor is examined in the order in which the processors were created. The job in

question is assigned to the first processor found that has a schedule that can accommodate the job and maintain a feasible schedule. If none of the processors' schedules can accommodate the job to be scheduled, another processor is created and the job assigned to that processor. When a job is assigned to a processor which is not empty, the first active interval is scheduled to start immediately following the completion of an execution interval of the first job that was assigned to that processor.

Within the constraints defined previously, Soh was able to prove that the algorithm based on the FDP assignment produced a schedule that required the minimal number of processors.

Since the binary frequency constraint is a rather strict requirement, Soh also considered schedules for job sets with arbitrary frequency distributions. For job sets with arbitrary frequency distributions, Soh developed four heuristic algorithms for determining schedules that required the minimal number of processors.

Each heuristic algorithm investigated by Soh was used to determine a multiprocessor schedule for thirty sets of periodic jobs, twenty-eight of which had thirty jobs and two with twenty jobs. The particular heuristic used for each algorithm was one of three types: (1) the heuristic ordered the jobs within the set and assigned the jobs to existing processors or created new processors based on the order of the job set; or (2) the jobs were assigned strictly by their ordering within the job set (nonincreasing frequency), but each job was assigned to a specific processor based on the effect its assignment would have on a given metric used to measure the "capacity" of a given processor schedule; or (3) a combination of a specific ordering other than

nonincreasing frequency of the jobs and an attempt to optimize a "capacity" metric.

The results of employing these algorithms on the given job sets was inconclusive. None of the algorithms could be termed "best" in most cases; and no algorithm could be defined which would be more advantageous than another based on the characteristics of the job sets. In addition, bounds were not established to relate the relative optimality of these algorithms to the actual minimal number of processors required for a job set.

The authors whose research we have discussed are the only ones, to our knowledge, who have published papers that specifically address the problem of multiprocessor scheduling of periodic tasks.

Except for the algorithms defined by Soh for job sets with binary frequency distribution and Dabholkar for job sets with integer frequency distribution and dynamic scheduling of jobs with equal execution times, there have been no results in the area of defining optimal schedules for periodic tasks. In particular, optimal schedules of job sets with arbitrary frequency distribution and arbitrary execution times have not been investigated. It is this problem that we will address in this investigation.

1.4 Problem Statement and Approach

In this section is informally described the specific scheduling problem with which we will be concerned. In addition, the rationale for our assumptions and an overview of the remaining chapters is presented.

The objective of this investigation is to define an algorithm for determining a valid nonpreemptive schedule for a given set of

periodic tasks with arbitrary request periods that requires the minimal number of identical processors. This algorithm will be optimal in the sense that if there exists a valid schedule, the algorithm will find a schedule with the minimal number of processors required for that job set.

Although there have been no published results that address the complexity of scheduling periodic tasks, in light of previously published findings concerning the complexity of the general scheduling problem, it seems unlikely that an efficient algorithm exists for scheduling periodic tasks except in special cases. Despite the fact the number of jobs in the job set are finite and the possible schedules must therefore be finite, the number of possible schedules may become prohibitively large as the number of jobs in the set increases. A question naturally arises concerning the justification of seeking an exact solution to a possibly NP-Complete problem.

For the specific scheduling environment with which we are concerned, the schedules are to be created once and for all - except for those that result from engineering changes to the system and its requirements. The schedules will not change dynamically within the operational system, and can therefore be created off-line. In addition, relative to the computational resources required to execute the tasks within the operational system during any given period of operation, the resources required to schedule the job set will be relatively small.

Furthermore, in some applications, the exact solution may be the only acceptable solution. For example, in a degraded mode of operation it may be essential that the minimal processor schedule be known to prevent catastrophic system failure. Or, when many copies of a system

are to be produced, the economics of the situation may require the minimal number of processors.

Finally, the exact solution of the problem would give insight into the characteristics of the problem which may aid in the development of heuristic algorithms and establish bounds against which heuristic algorithms can be compared.

Although the algorithm developed by Soh for scheduling job sets with a binary frequency distribution is optimal, there are many environments in which the job sets don't have a binary frequency distribution and cannot be converted to a binary frequency distribution. Moreover, the frequency of computation of each task in the operational system should be determined by the characteristics of the function the task performs and the accuracy requirements, not the scheduling algorithm constraints.

Despite the general results of scheduling theory which indicate that preemptive algorithms require fewer processors for the same job sets than do nonpreemptive algorithms, there are reasons for choosing a nonpreemptive algorithm. First, the preemption of tasks introduces unnecessary complexity into the executive or control structure of each processor and therefore reduces the computational resources available to perform the functional requirements. It also makes validation of the system software difficult and the definition of system operation with respect to the interrupt structure difficult to test. Although we will assume that any system overhead required for task initiation, completion, etc. is negligible, this is an assumption that is generally made for all scheduling algorithms, including preemptive algorithms which also have an inherent overhead due to the interrupt process.

Furthermore, the request rate for any given job cannot be assured in a preemptive environment (this is also true of nonpreemptive scheduling in which the deadline exceeds the execution time of the job). That is, although the job may complete the execution corresponding to each request prior to a given deadline, a preemptive algorithm can only guarantee that the execution corresponding to a given request will not complete more than the amount of the deadline after the request, but it cannot guarantee that the results of the execution of that task will be updated within a unit of time equal to the request rate. For example, if a preemptive environment is to make any sense, the deadline for the completion for each task's execution must be greater than the execution time of the task and not greater than the period of that task, i.e., for task j_i, $E_i < d_i < T_i$.

Suppose that the deadline associated with a job j_i is equal to T_i - the next request for execution of j_i. Then in a preemptive environment, a schedule of j_i might have a time history as illustrated below.

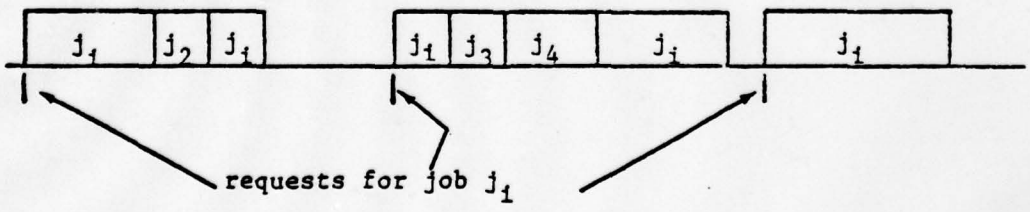


Figure 1-5 A Preemptive Schedule for j_i

With the deadline equal to T_i, it can only be guaranteed that the time between the completion of two adjacent execution intervals of j_i will not exceed $2T_i - E_i$ and will not be less than E_i. Hence you must allow at least one unit sample delay within any control loops of a system which uses preemptive scheduling.

Finally, in computer systems it would be difficult to have any task that isn't required to execute for at least one central processor unit (cpu) clock cycle. For multiprocessor systems, some synchronization requirement or a common clock would result in essentially the same restriction. Thus, in a digitized world, tasks are "preemptive" only in the sense of being divisible into a sequence of smaller nonpreemptive tasks; and nonpreemptive scheduling can be considered as the basic problem to be attacked for "real world" applications.

This then is our rationale for pursuing an optimal algorithm for nonpreemtive scheduling of periodic tasks with arbitrary frequency distribution. Toward this goal, we will exploit the characteristics of this specific problem to develop an effective means of enumeration of possible job schedules.

We begin in Chapter 2 by developing techniques of classifying jobs of a given job set into collections of jobs which do not exclude one another from a valid schedule on a uniprocessor. This classification is based entirely on the parameters which define the jobs - their periods and execution times. We further extend these results to define necessary conditions for the inclusion of any pair of jobs in a valid schedule on a single processor. We then examine the possibility

of the existence of a valid schedule for larger collections of jobs on a uniprocessor.

In Chapter 3, we define a formulation of the uniprocessor scheduling problem as a mixed integer programming problem, and define equivalence classes of valid schedules for a given subset of the job set. We then incorporate these results into an algorithm for the implicit enumeration of possible schedules to determine if a given subset of jobs does in fact have a valid schedule on a single processor.

In Chapter 4, we use the results of Chapters 2 and 3 to establish a lower bound on the number of processors required for a valid schedule of the job set. This bound is based on the characteristics of the jobs in the set and effectively reduces the number of possible multiprocessor schedules which must be examined.

A lattice structure is then defined that includes all of the possible valid schedules of the job set, and an algorithm defined that traverses the lattice structure. The algorithm determines successively tighter upper and lower bounds on the number of processors required until a schedule is determined which requires the minimal number of processors, or the search is terminated with a schedule that requires a number of processors that is within known worst case bounds relative to the optimal.

The applicability of the algorithms developed in the previous four chapters to the problem of scheduling sets of jobs which don't have an empty partial order is discussed in Chapter 5. The additions necessary to enable the algorithm to construct the minimal processor schedule for these sets of jobs is presented and illustrated with an appropriate example.

Finally in Chapter 6, we summarize the results of the investigation and discuss possible extensions.

CHAPTER 2

CLASSIFICATION OF PERIODIC TASKS

2.1 Introduction

In the preceding chapter we informally introduced the problem of nonpreemptive scheduling of periodic tasks in a hard real-time environment. A more formal definition of the problem will now be developed and the terminology that will be used in formulating a solution established.

The "brute force" technique of determining a minimum processor schedule for a set of independent tasks with the previous constraints would involve successively examining all possible permutations of the set of tasks on a processor set of first one, then two, then three and so on, until a schedule is found in which there is no conflict among task execution intervals. Of course, while the "brute force" method in general may be as effective as any other technique, the advantages of a more efficient technique, if it exists, drives us to try to design something that is at least " a little bit" better.

In this chapter we establish the terminology that will be used and more formally define the problem of determining a schedule that requires the minimum number of processors for a given set of tasks. In addition, criteria for the existence of a valid schedule of a subset of tasks on a single processor will be examined. The criteria will be based on the elements that define each job's time history and the total computational load of the jobs on any processor.

2.2 Schedules of Periodic Tasks

In this section the physical limitations of the problem and the assumptions which have been made relative to this scheduling environment will be discussed. We will also formalize these assumptions and the requirements of the framework of our particular problem.

Simply stated, the objective is to determine a valid schedule for a nonempty set of jobs $K=\{j_1, j_2, \ldots, j_n\}$ that requires a minimum number of processors. The job set K will be assumed to contain only periodic jobs each of which will require execution throughout the entire time period of the system's operation. Although there exists environments in which there are periodic tasks that require execution for only a small interval of the total time of the system operation and the requests for computation may not occur at system start, in order to guarantee that such periodic tasks will have access to the system resources when required and will not conflict with any other task, we will schedule these jobs as if they required computational resources throughout the system's period of operation.

We will assume that there is defined with the job set K a precedence relation $\prec$; such that for each j_i and j_k in K, if $j_i \prec j_k$, then the computations performed by the job j_k are dependent on the computations of the job j_i ; and j_i must precede j_k in any schedule. No assumption will be made as to a relationship between the frequencies of the two jobs, although in practice the jobs would most likely have periods which are related by an integer multiplier.

We will also assume that associated with each job j_i in the job set K is an ordered pair $(1/T_i, E_i)$ where T_i is a positive integer that represents the period of computation of the job j_i and that E_i is a positive real number that is the execution time required for each occurrence of the job j_i. We justify the choice of an integer period for the periodic tasks by considering the relationship of the task execution to the basic cycle time of a processor. Each computation of any job in the set will necessarily be constrained to occur at some integral multiple of the processor's basic cycle time. We could also so constrain the execution time of each task, but for the sake of generality will not do so at this time.

The relation E_i/T_i associated with the job j_i is a positive real number which we will call the load factor of j_i. The load factor of j_i represents the proportion of the available computational facilities of a single processor that are required by the job j_i. We will require that for each job j_i in the job set K, the load factor E_i/T_i must not exceed unity. That is, a given single processor must be capable of executing any job j_i from the job set K without any conlict occuring.

A valid schedule for a job set K is defined to be a sequence of functions $F=<P_i>$ such that, $P_i : K \longrightarrow t$, where t represents time and is given by $t=[0, \infty)$ and each function P_i satisfies the properties

discussed in the following paragraphs. Simply stated, each function P_i assigns a starting time for execution for each job in its domain. The total number of functions P_i in the sequence F is restricted to the range 1 to n, where there is associated with each function a processor. We will assume that the set of available processors is homogeneous. That is, each processor in the set is equally capable of performing the computations required of any of the jobs in K. For a schedule to be valid, there must be a subset of processors from the set of available processors which can fulfill the requirements specified in the constraints. This will require at least one processor since the job set K must be nonempty; and, since no job may require more computational power than is available on a single processor, i.e. $E_i/T_i \leq 1$, no more than n processors would be required for a valid schedule of an n job set. We therefore restrict the number of elements in the sequence F, called the cardinallity of F and denoted by $|F|$, to be $1 \leq |F| \leq n$.

For each function P_i and the subset of jobs that forms its domain, $D(P_i)$, we are further constrained so that for all j_k contained in the domain $D(P_i)$ the total load factor must not exceed unity. That is, the total load factor ($\sum_{k \text{ in } n_i} E_k/T_k$, where k is the index of the job j_k of the job set K, j_k is in the $D(P_i)$, and n_i is the set of all indicies of jobs in $D(P_i)$, for each processor P_i must be less than or equal to the total processing power available for each processor. We

further constrain the functions P_i, such that for a valid schedule only one processor within the set of all processors may have a schedule that contains a given job from the job set K. Each job may be assigned to one and only one processor. Thus for every function P_i and P_k where i and k are not equal,

$$D(P_i) \cap D(P_k) = 0$$

and

$$\bigcup_{q \in N} D(P_q) = K$$

where N is the set of indicies of the processors in a given schedule.

If we define the set of active tasks $A(t) = \{j \in K_i : P_q(j_i) \leq t < P_q(j_i) + E_i \text{ for all processors } q \in N\}$ to be the set of all tasks from the job set K that are active on any of the processors at any time t, at every instance of time t the cardinality of A(t) is constrained to be less than or equal to the cardinality of the sequence of partial functions F. Further, for each time t and each P_i,

$$P_i(j_k) \neq P_i(j_m) \quad \text{for all k,m in [1,..n], } k \neq m.$$

That is, there can be no more than one job scheduled to be active on any one processor at any time t.

Finally, if there exists a precedence relation between two jobs, then the predecessor must have an initial start time that precedes that

of its successor by an amount equal to the predecessor's execution time when the two jobs are scheduled on the same processor; i.e. for each j_i, j_k in K, if $j_i \prec j_k$, then $P_m(j_i)+E_i \leq P_m(j_k)$ for a given processor P_m. We will address the problem of assigning related jobs to different processors in Chapter 5.

The minimal processor schedule of a given set of periodic jobs, K, results in the minimal cardinality for the sequence F such that none of the previous constraints are violated.

We will assume that the sets of jobs to be scheduled are ordered in nonincreasing lexicographic order of the pair $(1/T_i, E_i)$. A lexicographic ordering of a set of vectors is an ordering on the first element of the vector; if there is a tie then an ordering on the second element and so on until every element has been considered. The relative size of the vector components determines the relationship between the vectors (Saa). For example, (0.25,0.1) is larger than either (0.1,0.1) or (0.25,0.05); and any job set with three jobs with frequency and execution time pairs equal to these would be ordered as follows: (0.25,0.1), (0.25,0.05), (0.1,0.1).

The requirements that we have placed on the scheduling of sets of periodic tasks ensures that once a given job begins its cyclic operation the active intervals will repeat in a predetermined fashion. In fact, we can define the job time sequence for any job by specifying the time of occurrence of its first active interval, t_{0i}, a nonnegative real number. The job time sequence for a job j_i is a linearly ordered

sequence of ordered pairs (t_I, t_C), where t_I is the time of initiation of any active interval of j_i and t_C is the time of completion of that active interval.

The sequence of initiation times for a job j_i is given by $t_{0i} + mT_i$ where m in [0,1,2,..); and t_{0i} is the start time of the initial active interval of j_i and will be called the job start time for brevity. The sequence of completion times for the same job is given by the set of times $t_{0i} + mT_i + E_i$ for m in [0,1,2,...). The sequence of ordered pairs for a job j_i is therefore $\langle (t_{0i} + mT_i , t_{0i} + mT_i + E_i) \rangle$ m in [0,1,...). Hence, we can define the job time sequence of any job j_i by the time of occurrence of its initial active interval; and we can abbreviate j_i's time sequence, and thus its time history, with $j_i(t_{0i})$.

Thus for any collection of jobs, there is no conflict, or a valid schedule exists on a single processor, if there exists no time, t, such that there are two or more integers p and q where

$$t_{0p} + mT_p \leq t < t_{0p} + mT_p + E_p$$

and

$$t_{0q} + rT_q \leq t < t_{0q} + rT_q + E_q$$

where m and r are any nonnegative integers; and p and q are indicies of jobs from the collection of jobs. In words, there will be no conflict if there is no time, t, that is contained within the active interval of two or more jobs.

For a given processor P_i and a job set K, there exists a processor job schedule, J_i, such that

$$J_i = \{j_m(t_{0m}) \mid j \text{ in } K \text{ and for the job time sequences defined by the set of job start times } \{t_{0m}\}, \text{ there is no conflict on } P_i\}$$

where t_{0m} represents the start time of j_m in this particular schedule. Trivially, an empty processor schedule is a processor job set for a processor with no tasks assigned. It is an empty set.

A processor time sequence is a partially ordered sequence of ordered pairs of job initiation and completion times for the m jobs scheduled on that particular processor. It describes completely the time history of a specific processor job schedule. For a processor job set of m jobs, the processor time sequence is made up of m linearly ordered subsequences of ordered pairs of individual job initiation and completion times.

The order in which the elements appear in a given processor job schedule $J_i = \{j_m(t_{0m}), j_k(t_{0k}), \ldots, j_r(t_{0r})\}$ defines the order in which the jobs were assigned to the processor P_i. Thus the indicies m,k,r of the particular jobs in a given processor job schedule, or processor schedule for shorter notation, need not be sequential. The indicies are the indicies of the job set K and the order in which the jobs occur within the processor schedule is a function of the scheduling algorithm. If the algorithm for scheduling jobs is order independent, the order of the elements in the processor job schedule is arbitrary.

Likewise, the initial start time of the jobs' first active intervals need not be sequential if the scheduling algorithm does not so restrict the scheduling of the individual tasks. For example, in the above processor schedule, t_{0r} may be less than t_{0m} indicating that, when scheduled, j_r's first active interval occured prior to j_m's first active interval.

A job schedule on a given processor is said to be a <u>compact schedule</u> if every task in that schedule has an initial active period that starts immediately following the completion of an active period of <u>one</u> of the other tasks assigned to that processor.

A job j_i is said to be <u>compact with respect to</u> a job j_r if j_i's initial active interval immediately follows one of job j_r's active intervals.

Further, a processor job schedule is said to be <u>compact in the initial interval</u> if every job in the processor job set is compact with respect to its predecessor in the job schedule, and the initial active period of each job in the schedule starts before the second active interval of the first job in the processor schedule. That is, the sequence of job initiation times within the processor job set, $t_{0i},..,t_{0m}$, are linearly ordered and bounded above by $t_{0i}+T_i-E_m$, where T_i is the period of the first job and E_m is the execution time of the last job scheduled.

We will abbreviate the processor job schedule J_i =

$\{j_k(t_{k0_m}), j_k(t_{k0_k}), \ldots j_r(t_{r0_r})\}$ with $J_i = \{j_m, j_k, \ldots, j_r\}$ if the schedule for the processor P_i is compact in the initial interval. In addition, we will abbreviate processor job schedules with a combination of implied initiation times and explicit initiation times. If a job is compactly scheduled with respect to the job immediately preceding it in the processor job schedule, the implied initiation time notation will be used. Otherwise, the initiation time will be explicitly defined for the jobs that are not compact with respect to the preceeding job in the schedule.

2.3 Restricted Regions for Periodic Tasks

We will begin our examination of classification strategies for periodic tasks by examining, first intuitively and then more formally, the instances in which two jobs can not possibly be scheduled on a single processor without a conflict occuring. We will make use of the resulting classification strategy in an attempt to reduce the number of possible combinations of jobs that must be considered to determine if a valid schedule exists for a given set of jobs. Without some means of classification, we would be faced with the task of analyzing possibly <u>all</u> combinations of jobs for a given job set which would in general be prohibitive.

Consider an arbitrary pair of jobs j_i and j_k such that $T_i \leq T_k$. We will arbitrarily choose the job j_i as a reference job and assume that the start time of its initial active interval is zero.

We will examine the feasibility of a processor schedule

$J=\{j_i, j_k(t_{0k})\}$ where t_{0k} must be greater than or equal to the execution time of j_i, E_i.

We observe that the cycles of j_i that would occur within any schedule with j_k, establish bounds on the start times of each active interval of j_k relative to the active intervals of j_i. No valid schedule can exist for these two jobs on the same processor if there is a conflict of any of j_k's active intervals with any of the active intervals of j_i. In other words, no active interval of j_k can begin less than E_k time units prior to the start of one of the active intervals of j_i; and, of course, no active interval of j_k can start sooner than E_i time units after the start of any of j_i's active intervals. Figure 2-1 illustrates regions of the time domain in which an active interval of j_k cannot start without interfering with an active interval of j_i. These regions contain the active intervals of the job j_i. Likewise, if the time history of the job j_k were examined, there would occur regions that contained the active intervals of j_k within which a conflict would occur if j_i were to start an active interval there.

The regions shown are determined by the period of j_i and the

execution times of both j_i and j_k. These regions contain the active intervals of the job j_i and are therefore periodic with a period equal to T_i and are $E_i + E_k$ time units in width. Thus we say that a <u>restricted region</u> of j_k relative to j_i is any time interval within the job j_i's time schedule during which j_k cannot initiate an active interval without conflicting with an active interval of j_i. It is equally true that the relationships of the job j_i and the job j_k could be reversed such that the above condition defines restricted regions of j_i relative to j_k. We will specify each restricted region by its upper bound, t_u, and its lower bound, t_l, where t_u and t_l are the time occurrences within the schedule such that job j_k, for instance, cannot begin an active interval between t_l and t_u with respect to j_i without a conflict occuring. The boundaries of the restricted regions are not contained in the restricted region; and an active interval can begin at t_u or t_l (as long as they are not contained in some other restricted region) without a conflict occuring. The width of each of the restricted regions that contain an active interval is equal to the sum of the execution time of the two jobs j_i and j_k. Note, restricted regions are not associated with a <u>single</u> job but rather with a pair of jobs. The restriction arises as a result of interaction of the two jobs. We will define each restricted

region, r_{ik} of j_i and j_k, by an ordered pair of non-negative real numbers (t_l, t_u), where t_l and t_u are the lower and upper bounds of that restricted region respectively. The set of all restricted reqions of j_i relative to job j_k, R_{ik}, is defined to be a set of ordered pairs of non-negative real numbers $R_{ik} = \{r_{ik} : r_{ik} \geq 0$ and $r_{ik} = (t_l, t_u)$ for all restricted regions of j_i relative to $j_k\}$. The above definition of a restricted region does not constrain restricted regions to contain an active interval although all those illustrated previously do contain an active interval of one of the jobs. In a later development, we will show that there do exist restricted regions which do not contain an active interval of one of the jobs to be scheduled.

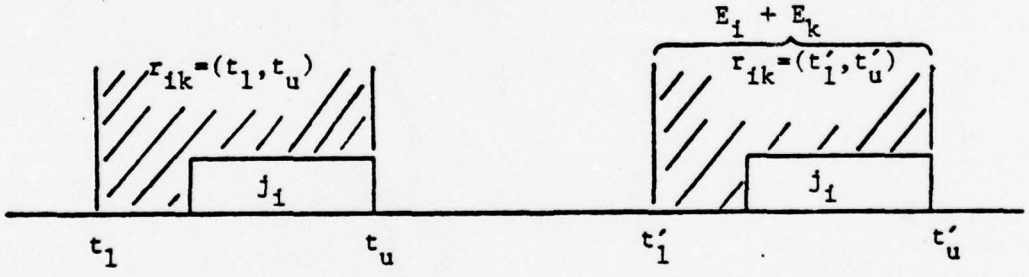


Figure 2-1 Restricted Regions That Contain Active Intervals

We will now examine in more detail the effect of the above illustrated restricted regions on schedules of periodic tasks. Let us consider an arbitrary pair of jobs j_i and j_k such that $T_k \geq T_i$. Without loss of generality, we can assume that for any pair of jobs we can arbitrarily assign a start time of zero to one of those jobs. This is a

result of the translation invariance of job schedules. A schedule does not depend on the name we give to the start time but rather the topological order properties of the job active intervals.

We are then able to view the time histories of the two jobs by successively examining "time frames" of the job with the zero start time, the reference job. A "time frame" of the reference job is the time interval between the start of one active period of the reference job and the start of the next active interval of that job. This is in part possible because of our requirement that the period of each job must remain constant throughout any schedule.

As an example, for the reference job j_1 with period T_1, we can successively examine the time intervals

$0 - T_1$

$T_1 - T_2$

$2T_1 - 3T_1$

and so on.

Every active interval of any other job which is to be scheduled with j_1 must occur some time within one of the times frames of j_1.

We can extend this concept to any pair of arbitrary jobs j_i and j_k as introduced previously. Thus it is easy to see that no conflict will occur between jobs j_i and j_k only if within each period of the reference job, j_k, no active interval of the non-reference job j_i is scheduled to begin execution sooner than the time of completion of the

active interval of the reference job; and no active interval of the non-reference job is scheduled to begin execution later than the length of its execution time prior to the start of the execution of the next active interval of the reference job.

For example, for the reference job j_k, the start of the active intervals of the job j_i that are to occur within the reference frame of nT_k to $(n+1)T_k$ of job j_k must not start prior to $nT_k + E_k$ nor after $(n+1)T_k - E_i$ as is illustrated by the restricted regions (cross-hatched regions) in the figure below.

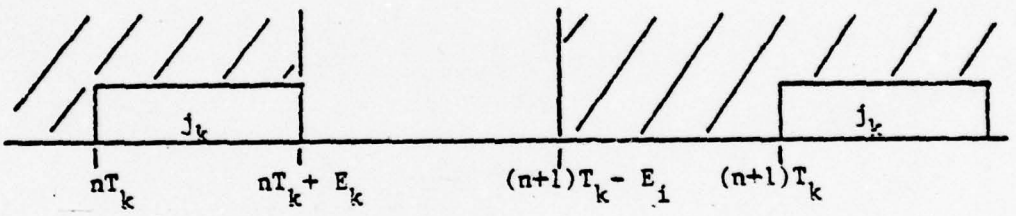


Figure 2-2

The $n+1^{st}$ Reference Frame of Job j_k

This restriction on the start times of the active intervals of the job j_i must not be violated within any of the reference frames of the job j_k if a valid schedule is to exists for j_i and j_k. So we can say that the two jobs j_i and j_k will not conflict only if every active interval of j_i starts no sooner than $nT_k + E_k$ and no later than $(n+1)T_k - E_i$ for <u>all</u> integer n greater than or equal to zero. That is, for <u>any</u> reference frame of the reference job j_k.

We can represent the conditional requirement for no conflict within a schedule mathematically as follows:

A given active interval, say the $(m+1)^{st}$, of the job j_i will occur within some reference frame, the $(N+1)^{st}$, of the job j_k. We define the integer N to be a function of the integer m;

$$N(m)=\text{Int}[(t_{0i}+mT_i)/T_k]$$

where Int[A] is the greatest integer less than or equal to the real number A.

Depending on the periods of the two jobs there may occur within any one interval of the job j_k more than one active interval of j_i. But, there will be no conflict of the $(m^*+1)^{st}$ active interval of the job j_i, where m^* is a nonnegative integer, within the $(N(m^*)+1)^{st}$ reference frame of j_k if the following inequality is true:

$$N(m^*)T_k+E_k \leq t_{0i}+m^*T_i \leq (N(m^*)+1)T_k-E_i$$

Furthermore, there will be no conflict of the job j_i and j_k at <u>any</u> time within a given schedule if the inequality

$$N(m)T_k+E_k \leq t_{0i}+mT_i \leq (N(m)+1)T_k-E_i$$

is satisfied for <u>all</u> integer $m \geq 0$.

Figuratively, we can examine the entire time history of any schedule containing the two jobs by overlaying "snapshots" or "folding over" one reference frame of j_k on top of the other for all time; and since any occurence of the two jobs' active intervals must occur within the reference frames, any conflict which might occur would be evident in one of the "snapshots" of the reference frame.

We perform the "folding over" process mathematically by examining the inequality above modulo the period of the reference frame - which, for our purposes, is the width of the "snapshot" we will be examining. It is this concept of "folding over" that we will use to establish a necessary condition for the existence of a valid schedule for an arbitrary pair of periodic jobs.

Given the set of inequalities

$$\{N(m)T_k + E_k \leq t_{0i} + mT_I \leq (N(m)+1)T_k - E_i \text{ for}$$

all integer $m \geq 0\}$

we "fold" the reference frame over by expressing the inequalities in the set modulo the period of the job j_k, T_k. That is,

$$\{((Nm)T_k + E_k) \bmod T_k \leq (t_{0i} + mT_I) \bmod T_k \leq ((N(m)+1)T_k - E_i) \bmod T_k$$

for all integer $m \geq 0\}$.

Since a requirement for any schedule to exist is that $E_k < T_k$ if

the load factor of j_k is to be less than one and $E_i < T_k$ if the idle interval of j_k is to possibly contain an active interval of j_i, this set of inequalities in turn becomes

$$\{E_k \leq (t_{0i} + mT_i) \bmod T_k \leq T_k - E_i \text{ for all integer } m \geq 0\}$$

once the specified modulo operation is performed on the elements of the inequality.

In the process of searching for a valid schedule for any pair of jobs we could examine the above set of inequalities for given start times for j_i until either no valid schedule is found or some t_{0i} determined that satisfies the inequalities for all m. This approach would of course be very time consuming if not impossible to complete for every pair of jobs in a job set. We prefer that there exist a more concise statement of the necessary condition given by the set of inequalities. Toward this end we will now show that it is only necessary to examine a simple relation of the periods and execution times of any pair of periodic tasks to determine if it is possible for any valid schedule to exist for the job pair on a single processor.

Specifically, we will show that no valid schedule exists for any pair of jobs j_i and j_k on a single processor unless the sum of the execution times of the two jobs does not exceed the greatest common divisor of the jobs' periods.

The <u>greatest common divisor</u> of given integers A and B is defined as follows:

Given integers x and y such that x=Xd and y=Yd where X, Y, and d are integers, then d is called a common divisor of x and y. Given integer A and B, if there is an integer D such the D is a common divisor of A and B; and every common divisor of A and B is a divisor of D; then D is the greatest common divisor of A and B. We will designate D by D=gcd(A,B) (Ste).

For the integers 8 and 12, for example, both 2 and 4 are common divisors of 8 and 12 while 4 is the gcd(8,12).

In order to establish that

$$E_i + E_k \leq \gcd(T_i, T_k)$$

is a necessary condition for a valid schedule to exist for any pair of jobs j_i and j_k, we will make use of the theory of groups. For the reader unfamiliar with these concepts, we have included in Appendix B a short summary of definitions and group theoretic notation applicable to our proofs.

Once we have established that the above condition is necessary for a valid schedule to exist for any given pair of jobs, we will make use of this fact to characterize jobs in a job set by collecting these jobs into subsets that do not exclude a valid schedule based on the relationship of the execution times and periods defined above.

We will begin the determination of the necessary condition by examining in detail the set of inequalities

$$\{E_k \leq (t_{0i} + mT_i) \bmod T_k \leq T_k - E_i \text{ ; for all integer } m \geq 0\}$$

and by a series of intermediate results develop a proof of the above necessary condition for a valid schedule to exist for any pair of periodic jobs.

From the theory of residue classes and modular addition (Bob), we know that

$$(t_{0i} + mT_i) \bmod T_k = ((t_{0i} \bmod T_k) + (mT_i) \bmod T_k) \quad \text{for all integer } m \geq 0.$$

We will begin our development by examining each element of the right hand side of the above equation one at a time starting with $(mT_i) \bmod T_k$. We will then reformulate the set of inequalities and define the conditions that are applicable for all nonnegative integer m.

The set of integers $Z_{T_k} = \{0,1,\ldots,(T_k - 1)\}$ together with the binary operation $+^*$ on Z_{T_k} is defined to be the additive group of integers modulo T_k, $(Z_{T_k}, +^*)$. In fact, any set $Z_n = \{0,1,2,\ldots,(n-1)\}$ together with the binary operation $+^*$ on Z_n, such that for a and b in Z_n

$$a +^* b = \begin{cases} a+b & \text{if } a+b < n \\ a+b-n & \text{if } a+b \geq n \end{cases}$$

is a group $(Z_n, +^*)$ for any n. $(Z_n, +^*)$ is usually referred to as the

additive group of integers modulo n (Liu). Furthermore, $(Z_{T_k}, +^*)$ is a cyclic group and is an Abelian group under addition modulo T_k.

Lemma 2.1 Let $H=\{(mT_i) \bmod T_k$ for all integer $m \geq 0\}$, then $(H,+^*)$ is a subgroup of $(Z_{T_k},+^*)$.

Proof To show that H, a finite subset of Z_{T_k}, is a subgroup of $(Z_{T_k},+^*)$, we must show that $+^*$ is a closed operation on H, where $+^*$ represents addition modulo T_k (Liu2).

Given arbitrary elements $h_1=(mT_i) \bmod T_k$ and $h_2=(nT_i) \bmod T_k$ such that m and n are nonnegative integers, we must show the $h_1 +^* h_2$ is an element of H. This is equivalent to showing that $(h_1+h_2) \bmod T_k$ is an element of H.

Suppose also that N, M, p, and q are nonnegative integers, then h_1 contained in H implies $h_1=mT_i+MT_k$; and h_2 in H implies $h_2 = nT_i+NT_k$.

Therefore, $h_1+h_2 = (m+n)T_i + (M+N)T_k$, where $(m+n)T_i = pT_i+qT_k$.

Hence, $(h_1+h_2) \bmod T_k$ is an element of H. And, $H=\{h=(mT_i) \bmod T_k$ for all integer $m \geq 0\}$ and the binary operation $+^*$ form a subgroup of the group

$(Z_{T_k}^{*}, +)$. QED

Because $(Z_{T_k}^{*}, +)$ is a cyclic group and we know that a subgroup of a cyclic group is also a cyclic group, there exists at least one, possibly more than one, element h in H which generates H.

We represent the group H generated by the element h by $\langle h \rangle = H$. For an additive group the group is formed by successive modular addition of the generator h, e.g. h, h+h, h+h+h, ..until nh=h for some integer n.

We will now determine a generator for the set $H=\{(mT_i) \bmod T_k$ for all $m \geq 0\}$.

Lemma 2.2 The subgroup $H=\{h=(mT_i) \bmod T_k$ for all integer $m \geq 0\}$ is generated by $(\gcd(T_i, T_k)) \bmod T_k$.

Proof To show that $\gcd(T_i, T_k) \bmod T_k$ is a generator of H, we must show that $\gcd(T_i, T_k) \bmod T_k$ is an element of H and that every other element of H can be formed by successive modulo addition of $\gcd(T_i, T_k) \bmod T_k$ with itself.

Suppose $\gcd(T_i, T_k) = T_k$, then $\gcd(T_i, T_k) \bmod T_k = 0$. If $\gcd(T_i, T_k) = T_k$, then $T_i = T_k$ (since $T_i \leq T_k$) and there exists but a single element in the set H. That element is zero. Thus the set H contains $\gcd(T_i, T_k) \bmod T_k$ and it is generated by it.

On the other hand, suppose $\gcd(T_i, T_k) < T_k$, then $\gcd(T_i, T_k) = \gcd(T_i, T_k) \bmod T_k$.

We will now show that $\gcd(T_i, T_k)$ is an element of H.

From the Euclidean algorithm of number theory we know that there exist integers p and q such that $\gcd(T_i, T_k) = pT_k + qT_i$ (Ste).

The integers p and q are not unique, in fact, $\gcd(T_i, T_k) = (p+nT_k)T_k + (q-nT_k)T_i$ for every natural number n. Thus regardless of the integers p and q there exists a natural number n such that $p+nT_k$ is an integer greater than or equal to zero; and $\gcd(T_i, T_k) \bmod T_k = ((p+nT_k)T_k) \bmod T_k$. Hence, $\gcd(T_i, T_k)$ is an element of H.

We will now show that every element in H is generated by $\gcd(T_i, T_k)$.

The set $H = \{h : h = (mT_i) \bmod T_k \text{ for all integer } m \geq 0\}$ is equal to the set $\{h : h = mT_i - nT_k^* \text{ where } n^* = \mathrm{Int}[mT_i/T_k],\ m \geq 0\}$ where $\mathrm{Int}[mT_i/T_k]$ is the greatest integer less than or equal to mT_i/T_k.

From the theory of numbers, the equation $h = mT_i - nT_k^*$ can be recognized as a Diophantine Equation (Ste) (see Appendix A). There exists a solution to the Diophantine Equation $h = mT_i - nT_k^*$ if and only

if the integer h is divisible by $\gcd(T_i, T_k)$.(Ste) Thus every element of H must be zero, (since zero is defined to be divisible by any nonzero integer), or an integral multiple of $\gcd(T_i, T_k)$. Therefore the $\gcd(T_i, T_k)$ must be a generator of H since under modulo addition, every element of H is an integral multiple of $\gcd(T_i, T_k)$. QED

As a result of the above Lemma, we can express the set $\{(mT_i) \bmod T_k$, for all integer $m \geq 0\}$ as the set $\{\text{rgcd}(T_i, T_k); r \text{ in } Z_{n_k} = [0,1,..,(T_k/\gcd(T_i, T_k))-1]\}$

We will now make use of the above representation of the set H to determine for the original set of inequalities that formed the necessary conditions for a valid schedule the limits that are common to all of the inequalities in the set. We will show that for two jobs j_i and j_k, a necessary condition for a valid schedule is that the set of inequalities $\{E_k \leq (t_{0i}) \bmod \gcd(T_i, T_k) + \text{rgcd}(T_i, T_k) \leq T_k - E_i ; r \text{ in } Z_{n_k}\}$ be satisfied for all r.

We will define the operation of (A)modulo B, for any nonnegative real number A and any natural number B, to be equal to $A_f + (A_I) \bmod B$ where $A_f \geq 0$ is the fractional part of A and A_I is the integer portion whereby $A = A_f + A_I$.

The job start time of j_i can be expressed as the sum of an integer and fractional elements such that $t_{0i} = I + F$ where $0 \leq F < 1$ and I is a nonnegative integer.

$(t_{0i})\ \text{mod}\ T_k = (I+F)\ \text{mod}T_k = F + (I)\text{mod}T_k$. The integer $(I)\text{mod}T_k$ is an element of the group $(Z_T^*, +_k)$ so that

$$(I)\text{mod}T_k + (mT_i)\ \text{mod}T_k$$

and $(I)\text{mod}\ T_k$ must be some coset (see Appendix B) of the group generated by $\gcd(T_i, T_k)$ in the group $(Z_T^*, +_k)$ for any integer $m \geq 0$ and job start time t_{0i}.

The element $(I)\text{mod}\ T_k$ can therefore be represented by I^* $+_k$ $\gcd(T_i, T_k)$ where I^* is an integer such that $0 \leq I^* \leq \gcd(T_I, T_k)$ and k^* is a nonnegative integer.

We will now look again at the original set of inequalities $\{E_k \leq (t_{0i} + mT_i)\ \text{mod}T_k \leq T_k - E_i$; for all integer $m \geq 0\}$ and the decomposition of $(t_{0i} + mT_i)\ \text{mod}T_k$ into $(t_{0i}\ \text{mod}T_k + (mT_i)\ \text{mod}T_k)$.

First of all, the start time of job j_i, t_{0i}, is a real number greater than zero so that $t_{0i} = t_f + t_I$ where t_f represents a real number $0 \leq t_f < \gcd(T_i, T_k)$ and t_I is a nonnegative integer defined by $t_I = n' \gcd(T_i, T_k)$ where $n' = \text{Int}[t_{0i}/\gcd(T_i, T_k)]$.

The set $\{(t_{0i} + mT_i) \bmod T_k$ for all integer $m \geq 0\}$ is equal to $\{(t_{0i}) \bmod \gcd(T_i, T_k) + r \gcd(T_i, T_k)$ for all r in $[0,1,..,n_k]\}$.

The set of inequalities that define the necessary condition for a valid schedule therefore becomes $\{E_k \leq (t_{0i}) \bmod \gcd(T_i, T_k) + r\gcd(T_i, T_k) \leq T_k - E_i$ for all $r \in Z_{n_k}\}$

We are finally prepared to prove the most important result of this chapter. We will show at this time that a necessary condition for any valid schedule to exist for the jobs j_i and j_k on the same processor is that $E_i + E_k \leq \gcd(T_i, T_k)$.

Theorem 2.1 There exists a valid schedule on a single processor for the jobs j_i and j_k of the job set K only if $E_i + E_k \leq \gcd(T_i, T_k)$.

Proof We previously demonstrated that a valid schedule for j_i and j_k can exist only if the set of inequalities $\{E_k \leq (t_{0i} + mT_i) \bmod T_k \leq T_k - E_i$ for all integer $m \geq 0\}$ is always satisfied. Or equivalently,

it is necessary that the set of inequalities {$E_k \leq (t_{0i} \bmod \gcd(T_i, T_k) + r\gcd(T_i, T_k) \leq T_k - E_i$ for all $r \in Z_{n_k}$, given t_{0i}} always be satisfied.

For a given job start time t_{0i}, the quantity $(t_{0i}) \bmod \gcd(T_i, T_k)$ has a set of lower bounds defined by E_k and r. Likewise, there exists a set of upper bounds determined by T_k, E_i and r. The greatest lower bound is E_k, and the least upper bound is $T_k - E_i - (T_k - \gcd(T_i, T_k)) = \gcd(T_i, T_k) - E_i$.

Every one of the inequalities must satisfy the relation defined by the least upper bound and the greatest lower bound for the set of inequalities to be satisfied and, therefore, for a valid schedule to exist for j_i and j_k on the same processor.

That is,

$$E_k \leq \gcd(T_i, T_k) - E_i$$

or $E_i + E_k \leq \gcd(T_i, T_k)$ QED

This is the necessary condition we alluded to previously. By examining every pair of jobs within a job set we can determine if a valid schedule could possibly exist with the given pair of jobs on the same processor. It is therefore a relatively easy task to determine for every pair of jobs in a given job set those jobs that could possibly be

assigned to the same processor and, at the same time, those tasks that could never be scheduled on the same processor.

A <u>subschedule</u> J_p of a given valid schedule of n jobs $J=\{..,j_i(t_{0i}),...j_k(t_{0k}),..,j_m(t_{0m})...\}$ is any nonempty subset of J with n-1 or fewer elements. The schedule $J_p = \{j_i(t_{0i}),j_k(t_{0k}),j_m(t_{0m})\}$ is, for example, a three-job subschedule of the schedule J shown above. In the time domain, a subschedule of a given valid schedule J is formed by eliminating all of the active intervals of all of the jobs in the original schedule J except the active intervals of the jobs in the subschedule.

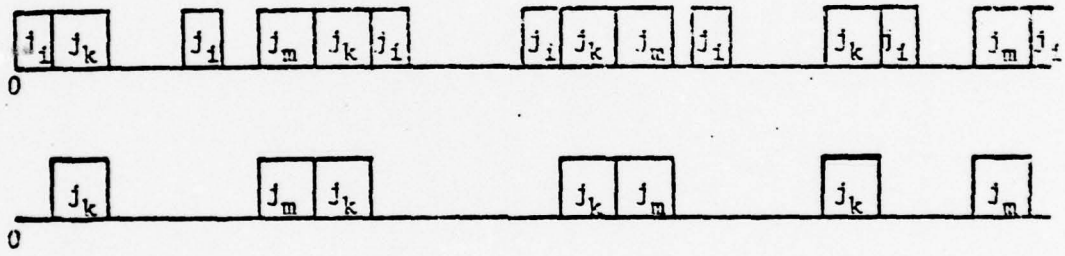


Figure 2-3 A Three Job Schedule and One Subschedule

With the concept of a subschedule in mind we will make the obvious extension of the above theorem to larger collections of jobs.

<u>Corollary 2.1</u> No valid schedule exists on a single processor for any collection of two or more jobs K_i from a job set K if there exists any pair of jobs j_i and j_k in K_i such that $E_i + E_k > \gcd(T_i, T_k)$.

<u>Proof</u> No valid schedule exists for j_i and j_k when assigned to

the same processor. Hence there will be no single processor schedule that is valid which contains both j_i and j_k since each subschedule with j_i and j_k can not be valid. QED

Prior to pursuing the development of additional techniques and consequences of classifying jobs from the job set K, we will examine in more detail the restricted reqions of a given pair of jobs j_i and j_k. We will do this in order to further develop concepts which we will use later to define possible schedules for periodic tasks and to gain further insight into the restrictions on determining job start times for valid schedules of periodic tasks.

2.4 Restricted Regions Revisited

In addition to the restricted regions shown above, the periodic nature of j_k and all of the jobs considered makes other restricted regions within the time schedule of j_i possible. These restricted regions result because when any active interval of j_k is scheduled to begin within one of these regions, not a resticted region that contains an active interval of j_i, there would eventually occur within the time schedule of j_i an active interval of j_k within one of the restricted regions that <u>does</u> contain one of the active intervals of j_i. These restricted regions of j_k relative to j_i occur at integral multiples of the period T_k from each of the restricted regions that contain active

intervals of j_i. For example, when job j_i has a schedule as shown below, a restricted region $r_{ik}(m)$ would result from a mapping of the restricted region $r_{ik}(n)$ back an amount equal to the job period T_k.

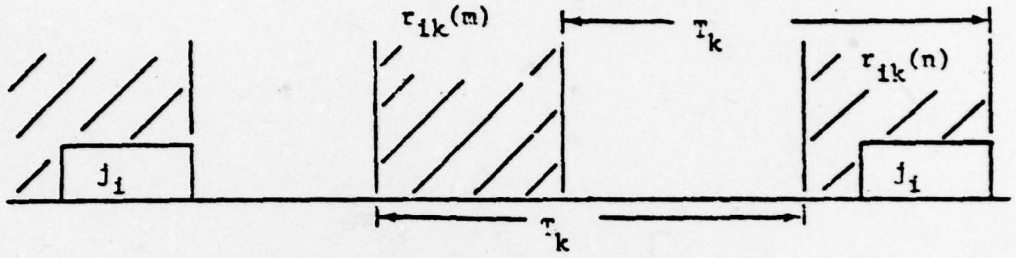


Figure 2-4 The Relationship of Restricted Region $r_{ik}(m)$ to $r_{ik}(n)$

Before we formalize a technique for generating all of the restricted regions of a given pair of jobs, we will show by example how restricted regions occur within a specific schedule of a job pair.

Consider the two jobs below and the resulting restricted regions that contain active intervals of j_1 (Figure 2-5).

Job	Period	Execution time
1	3	0.20
2	5	0.50

Figure 2-5 Restricted Regions That Contain Active Intervals of j_1

If, for example, j_2 were to have its initial active interval

begin at 2 units of time after t_{01}, there would be a conflict with j_1 at 12 units of time. Since for $j_2(2)$, j_2 would have a job time sequence

<2, 2.5, 7, 7.5, 12, 12.5,.....>; and

a conflict will occur at t=12 since there is an active period of j_1 already scheduled to occur there.

So, we know that t_{02}=2 is within some restricted region of j_k relative to j_1. In addition, the point t=7 is also in a restriced region since an active period of j_2 beginning at that time would conflict with j_1 at the same point, t=12, although we don't know the upper and lower bounds of these regions at this time. We do know that there exists an integral multiple of T_2 between the respective upper and lower boundaries of the restricted regions. Thus the restricted region that contains the schedule time t=2 is in the region that results from a mapping of the region which is defined by (11.5,12.2), an amount equal to $2T_2$=10. The boundaries of the region are (1.5,2.2). There is also a restricted region (not shown in Figure 2-6) defined by (6.5,7.2) since any active period of j_2 that starts during this time frame will also confict with j_1 at t=12. A restricted region may be defined by restricted regions which occur before it as well as after it in time. For example, the restricted region shown in Figure 2-6 as (7.5,8.2) is defined by restricted regions at both (2.5,3.2) and (17.5,18.2). Note,

that the latter is $3T_k$ units of time from the former.

The restricted regions that have been identified so far are shown in below.

Figure 2-6 Restricted Regions

In fact, any point within a schedule that will result in an active period of j_2 starting within any of the restricted regions shown in Figure 2-6 is also within a restricted region.

We will say that a restricted region, r(m), is generated by another restricted region, r(n), if an active interval of j_p which starts in r(m) will conflict with j_q in r(n).

The restricted regions generated by a given restricted region, say (5.5,6.2), occur with the same period as j_2. The sequence of lower bounds of the restricted regions generated by this restricted region is, for example, t={0.5,5.5,10.5,15.5,....}.

The first four restricted regions generated by the active interval at t=5.5 or containing that interval are:

$$0.5 < t < 1.2$$
$$5.5 < t < 6.2$$
$$10.5 < t < 11.2$$
$$15.5 < t < 16.2$$

In a similar, manner we can generate the bounds of the restricted regions which would result in conflict within one of the other original restricted regions shown in Figure 2-5.

All of the restricted regions generated in this way for the schedule $\{j_1, j_2(t_{02})\}$ are shown in Figure 2-7.

Figure 2-7 All Restricted Regions of j_2 Relative to Job j_1

Suppose instead, we considered the schedule $\{j_2, j_1(t_{01})\}$. The restricted regions that would result when the same technique as used above to generated restricted regions is applied to this schedule are shown below.

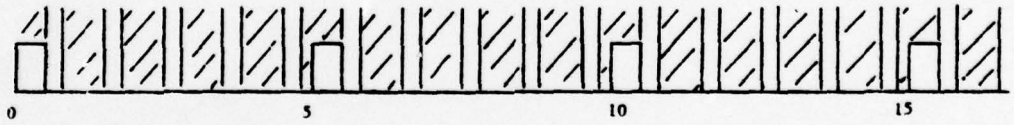


Figure 2-8 Restricted Regions of $\{j_2, j_1(t_{01})\}$

Note that, except for a bias that results from the difference in the two execution times, the restricted regions are exactly the same regardless of which job is used as the reference job (the job with an initial active interval start time assumed to be zero in this example); and relative to each other either job can effectively serve as the

reference job. Restricted regions, therefore, always exist with upper bounds equal to $t_{0i} + E_i$ and $t_{0k} + E_k$ of one job relative to the other.

Although the example above describes the generation of restricted regions only for schedules in which one of the jobs in the pair has a zero start time, the concept can be extended to any job pair within which no job is assumed to have a zero start time. In order to extend the concept of restricted regions to pairs of jobs which do not have a zero start time, we will reexamine the formation of the restricted regions. The only difference in the restricted regions shown above when compared with the restricted regions that would result if neither t_{01} or t_{02} is equal to zero, is the possibility of restricted regions of the nonreference job occuring prior to the first active interval of the reference job. By definition, t_{0i} is the first active interval of a job j_i within any processor schedule. So, since no job can begin at $t < 0$, no active period of j_i can occur prior to t_{0i}; and the restricted regions generated by a given active interval of j_i includes those regions that occur prior to that active interval. The determination of restricted regions for the case of no start time of any job in a pair equal to zero differs from the above example by only a bias factor equal to the start time of the reference job.

Since any conflict of two jobs occurs when one of the jobs is scheduled to begin execution within some restricted region relative to the other job, for a valid schedule no active interval of <u>either</u> job may begin within a restricted region relative to the other job of the pair.

We would therefore like a formal means of generating the restricted regions of a given job pair and assurance that every restricted region can be found.

We will demonstrate in the following theorem that there is a relatively simple method for constructing the restricted regions generated by the active intervals of the reference job of a given job pair. We will also show that there exists no other restricted regions within the schedule of an arbitrary pair of jobs.

Theorem 2.2 The set of restricted regions of a job j_k relative to a job j_i, which contain the active intervals of j_i or are generated by the restricted regions that contain the active intervals of j_i, is defined by the Diophantine Equation $pT_i + mT_k = y_u$, where y_u is an integer that defines upper boundary of each restricted region as a function of the integers p and m.

Proof The figure below illustrates again the active intervals of a job j_i and the restricted regions for j_k that contain each active interval of j_i.

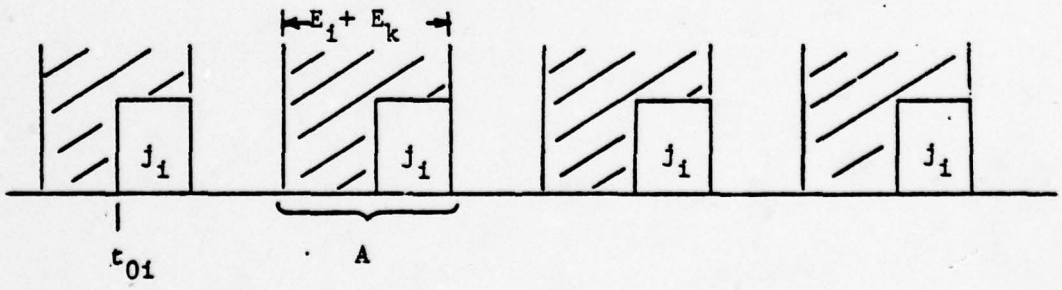


Figure 2-9 Restricted Regions for j_k Relative to Job j_i

We will now construct for each of the restricted regions of j_k that contains an active interval of j_i all of the restricted regions which they generate.

As defined previously, the restricted regions generated by a given restricted region are those intervals of the time domain for which a conflict of j_k would occur at the given active interval of j_i if any active period of j_k were to start within that time period. Because of the periodicity requirement for our particular class of jobs, the restricted regions so generated occur at integral multiples of the period of j_k, T_k, from the restricted region which generated them.

Consider the upper boundary of the restricted region "A" shown above. The time t that defines that boundary is $t_{0i} + T_i + E_i$. In fact, the upper boundary of every restricted region which contains an active interval of j_i is given by the equation $t = t_{u\,0i} + pT_i + E_i$ where p is an integer in the set [0,1,..).

The set of upper boundaries of the restricted regions generated by the upper boundary $t_{0i} + T_i + E_i$ is defined to be

{$t > 0$ $t = t_{u\,0i} + T_i + E_i + mT_k$, m an integer}.

In a similar manner, we can construct the set of upper boundaries of the restricted regions generated by <u>each</u> of the restricted regions which contain active intervals of j_i. In other words, the time of occurrence of the upper bound of each of the restricted regions is a

exists a solution to the Diophantine Equation n=mI - n I if and only
i k

function of the start time of the reference job, t_{0i}, the $(m+1)^{st}$

multiple of the period of the non-reference job, and the $(p+1)^{st}$ active interval of the reference job.

The set of all upper bounds of restricted regions (both those which contain active intervals , and those generated) of j_k relative to j_i is defined by the set $\{t_u > 0 \quad t_u = t_{0i} + pT_i - E_i + mT_k \; ; \; p$ a nonnegative integer, m an integer$\}$. The restricted regions so defined will each be $E_k + E_i$ in width and have lower bounds defined by $\{t_l \geq 0: \quad t_l = t_u - E_u - E_i$ for all p and m$\}$.

Every restricted region of the job j_k relative to j_i is therefore defined by the set of ordered pairs $R_{ik} = \{(t_l, t_u): t_u = t_{0i} + pT_i + mT_k + E_i$ and $t_l = t_u - E_u - E_i\}$ where p and m are as defined previously. Associated with each upper bound t_u is and integer $y_u = t_u - t_{0i} - E_i$ such that the upper bound of every restricted region is defined by the set $\{t_u : t_u = y_u + t_{0i} + E_i \; ; \; y_u = pT_i + mT_k\}$ for the same p and m.

The equation $y_u = pT_i + mT_k$ is a Diophantine Equation for which we know there exists a solution if and only if y_u is divisible by $\gcd(T_i, T_k)$. (Ste)

Since a conflict of j_k with j_i is possible only at the times

when the job j_i is in the process of execution (in an active interval), and the restricted regions generated by the two Diophantine equations for the upper and lower boundaries includes every active interval of j_i and each interval of the time domain in which the start of an active interval of j_k would at some time result in a conflict with j_i, the restricted regions so generated are the only restricted regions of j_k relative to j_i. QED

As a result of the above theorem, we know that we can determine each and every restricted region of job j_k relative to j_i by solving a Diophantine Equation and employing the characteristics of the two jobs. The following corollary establishes an easier means of determining the restricted regions for a job pair.

Corollary 2.2 For a given pair of jobs j_i and j_k, the restricted regions of j_k relative to j_i are given by $R_{ik} = \{(t_l, t_u): t_u = t_{0i} + E_i + m\gcd(T_i, T_k);\ 0 \leq t_l = t_u - E_i - E_k$; for all integer m$\}$.

Proof As there is a solution to the Diophantine equation if and only if $t_u - t_{0i} - E_i$ is divisible by $\gcd(T_i, T_k)$ and t_u will always be greater than zero, the upper bounds can be generated by considering integer multiples of $\gcd(T_i, T_k)$ for a given start time t_{0i}. QED

In the previous theorem it was shown that all of the restricted regions of a given job relative to a reference job were periodic with a

period equal to the greatest common divisor of the jobs' periods. Each of the restricted regions we know to have a width equal to the sum of the execution times of the two jobs. We can again see the relationship of the greatest common divisor of the jobs periods to the execution time sum as a limiting factor for a possible schedule with a given pair of jobs on the same processor - if the sum of the execution times exceeds the greatest common divisor of the jobs' periods the entire time domain is contained within restricted regions.

We will at this time continue our previous development of the concept of grouping subsets of jobs from the job set such that each element in a given subset will not exclude any other element from being assigned to the same processor.

2.5 Compatibility Classes of Periodic Tasks

The ability to form subsets of the job set such that none of the the jobs within any of the subsets prevents the others from being assigned to the same processor when considered pair wise makes it possible for us to form collections of subsets of jobs that may have a valid uniprocessor schedule. We will in this section develop a technique of forming the largest collections of jobs that may have a valid schedule on the same processor - that is the largest collections of jobs that do not exclude one another from being assigned to the same processor. Finally, in this section we will illustrate by means of an example how this technique works in practice and is implemented on a digital computer.

We define a binary relation C on a set of periodic jobs $K=\{j_1, j_2, \ldots, j_n\}$ as follows:

For any pair of jobs j_i and j_k contained in the set K, j_i is related to j_k by C, represented by $j_i C j_k$, if and only if there exists a valid schedule for j_i and j_k on a empty processor. That is, only if there is some schedule $J=\{j_i(t_{0i}), j_k(t_{0k})\}$.

From previous results when $i \neq k$, we know that the above relation is equivalent to the requirement that the sum of the execution times of j_i and j_k not exceed the greatest common divisor of periods. While for $i=k$, (i.e., one and the same job) as long as the execution time of j_i does not exceed its period (a requirement for all of the tasks being considered), each job can be scheduled on an empty processor and is therefore related to itself.

The relation C defined above is in fact a compatibility relation. That is, it is reflexive, symmetric, but not transitive; and as a result, will not in general partition the job set (Pra). But, the relation C will create subsets of the job set K which will catagorize the elements of K by collections of tasks that do not exclude each other, when considered pairwise, from inclusion in the same processor schedule.

Therefore, for any job set K, job j_i is compatible with job j_k if and only if $E_i + E_k \leq \gcd(T_i, T_k)$.

We can now determine for every pair of jobs, j_i and j_k, in a job set, K, whether j_i and j_k are compatible or incompatible by examining

the sum of the jobs' execution times and the greatest common divisor of the jobs' periods.

A subset K_r of the job set, K, will be said to form a compatibility class of K if for every pair of jobs, j_i and j_k, in K_r, $j_i C j_k$. That is j_i and j_k are compatible.

We will represent the compatibility relations for every pair of jobs in a job set K by a relation matrix R of C where each element r_{im} is defined as follows:

$$r_{im} = \begin{cases} 1 & \text{if } j_i C j_m \\ 0 & \text{if } j_i \text{ not compatible with } j_m \end{cases}$$

The relation matrix for any job set is a symmetric matrix ($r_{im} = r_{mi}$) with all of the elements on the main diagonal equal to unity. (Although the representation is the same as that used for restricted regions, the context of its use should make it clear which is meant.)

A maximal compatible K_i of jobs from a given job set K is a compatibility class of K, which will not remain a compatibility class if any job from K that is not in K_i is added to K_i. If there is a job in K that is compatible only with itself, then that job forms a maximal compatible with a single element. A maximal compatible is therefore a "largest" collection of a given subset of jobs from the job set that do

not exclude each other from assignment to a given processor. The collection of all of the maximal compatibles of a given job set will therefore represent all of the possible collections of jobs which do not exclude each other from a uniprocessor schedule. We will now develop a technique for forming all of the maximal compatibles of a given job set.

The problem of determining for a given set of elements the collection of maximal compatibles arises in other contexts. For example, in the theory of finite automata the minimization of incompletely specified machines uses maximal compatibles.

There are several ways of representing the problem and consequently several techniques for its solution. The problem can be represented as the determination of the collection of maximal complete subgraphs in a symmetric graph where the elements of the set are viewed as nodes of an undirected graph and the edges represent the existence of the compatibility relation between the nodes (Das). Equivalently, the set of all maximal compatibles can be determined by an algebraic manipulation of a Boolean equation of n variables where each variable represents a particular element of the set K and n is the cardinality of the set (Yan).

The algorithm presented here incorporates some of the terminology and techniques of each of the methods mentioned above to generate the collection of all maximal compatibles of a given job set K.

Previously we determined for a given job set the matrix that represented the compatibility of each pair of jobs in the set. We can just as readily represent by a matrix or a graph the condition opposite to that of compatibility of a pair of jobs - that of incompatibility. The pairwise incompatibility of the jobs of the job set can therefore be

represented by a matrix or a graph where the existence of a 1 in the matrix or an edge in the graph represent the incompatible relation. That is, if j_i is incompatible with j_k ($E_i + E_k > \gcd(T_i, T_k)$) then

$\bar{r}_{ik} = 1$, otherwise $\bar{r}_{ik} = 0$.

For ease of representation we can consider $j_1, j_2, \ldots, j_n$, the variables that represent the jobs in the job set, to be Boolean variables that represent the inclusion of each particular job in a given subset of the job set. If the job j_i is present in the subset, we will represent that fact by using j_i. If j_i is not in the subset, we use $\bar{j}_i$ to indicate this condition. For example, for $K=\{j_1, j_2, j_3, j_4\}$ the subset $K'=\{j_2, j_4\}$ could be represented by the Boolean product $\bar{j}_1 j_2 \bar{j}_3 j_4$. Thus the incompatibility graph of a given n-job set can be represented by a Boolean function of n variables j_1 through j_n in the form of a product of Boolean sums, $\ldots(j_i + j_k)(j_l + j_k + j_m)\ldots$ where j_i and j_k are connected by an edge in the incompatibility graph of the job set K as are j_l, j_k, and j_m - i.e. they are incompatible.

Employing the notation of Yang, we will represent every pair of incompatible jobs j_i and j_k as a Boolean sum, $(j_i + j_k)$, and then a Boolean function, f, of n variables j_1 thru j_n as a product of these

sums. The function f defines every pair of incompatible jobs in the set.

For example, for K above, j_1 is incompatible with j_2 and j_2 is incompatible with j_4 while all other jobs are pairwise compatible. The function $f(j_1,j_2,j_3,j_4) = (j_1+j_2)(j_2+j_4)$.

We can make use of the Shannon Expansion Theorem (Pra) to expand the Boolean function f into a boolean sum of products with respect to the Boolean variables $j_1,j_2,\ldots,j_n$ such that

$$f(j_1,j_2,\ldots,j_n) = j_i f(j_1,\ldots,j_{i-1},1,j_{i+1},\ldots,j_n) + \bar{j}_i f(j_1,\ldots,j_{i-1},0,j_{i+1},\ldots,j_n)$$

and each of the Boolean functions $f(j_1,\ldots,j_{i-1},1,j_{i+1},\ldots,j_n)$ and $f(j_1,\ldots,j_{i-1},0,j_{i+1},\ldots,j_n)$ are themselves Boolean functions of n-1 variables, where 0 and 1 are respectively the additive and multiplicative identity elements of a Boolean Algebra, and can further be expanded with respect to one of the remaining Boolean variables j_i (Pra). Successive expansions of each of the functions finally terminates in a Boolean polynomial of variables $\{j_1,\ldots,j_n\}$ and their complements with coefficients of $f(b_1,b_2,\ldots,b_n)$ where b_i is an element of

B={0,1}; and each coefficient of the Boolean polynomial is also an element of the set B (i.e., each coefficient will be either a logical 1 or 0). The Boolean function $f(j_1,\ldots,j_n)$ is thus represented by a Boolean sum of products that is in minterm canonical form (Pra). That is, <u>each</u> variable j_i of the set of variables $\{j_1,j_2,\ldots,j_n\}$ appears in <u>each</u> Boolean product term as either j_i or $\bar{j}_i$.

Since each product $\bar{j}_i j_k j_m \ldots j_p$ is covered by $j_k j_m \ldots j_p$ in the sum of products representation, each complemented variable $\bar{j}_i$ in any product can be deleted. So that, after the final expansion and deletion of complemented variables, the Boolean function $f(j_1,\ldots j_n)$ is represented in the form of a sum of products of uncomplemented variables j_i.

The algorithm using the Boolean algebra representation of the incompatible relations (Yan) implies that each product term of uncomplemented variables formed using the expansion is irredundant, (i.e., there is no product term in the expanded function which is implied by any other product term in the expansion) hence there is no need to look for redundancies among the solutions generated. Further, except for the absence of redundant solutions, this algorithm was shown to be equivalent to that of Das.

Das used a table of node indices to represent the existence of edges in the incompatibility graph. That is, for an edge in the graph,

say Q_1, a column in the edge table for that graph contains the two nodes that are joined by that edge. Each edge in the incompatibility graph is enumerated and included in the edge table as two node indices.

There is a one to one correspondence between the edge table as defined above and a binary incompatibility matrix in which a one in row i and column j defines an edge in the incompatibility graph from node i to node j, such that if j_i is incompatible with j_k there is a 1 in row i and column k and a 1 in row k and column i.

The expansion, by means of Shannon's Theorem, of the original n-variable Boolean function can be represented as a binary tree in which each node represents an m-variable Boolean function where m is contained in [0,1,...n]. Each edge in the tree has associated with it a Boolean variable j_i or $\bar{j}_i$ that defines the Boolean variable about which the previous Boolean function was being expanded; and whether the succeeding node is f(....,1,...) or f(....,0,...) respectively. For example, the Boolean function $f(x_1, x_2)$ can be represented by the tree below.

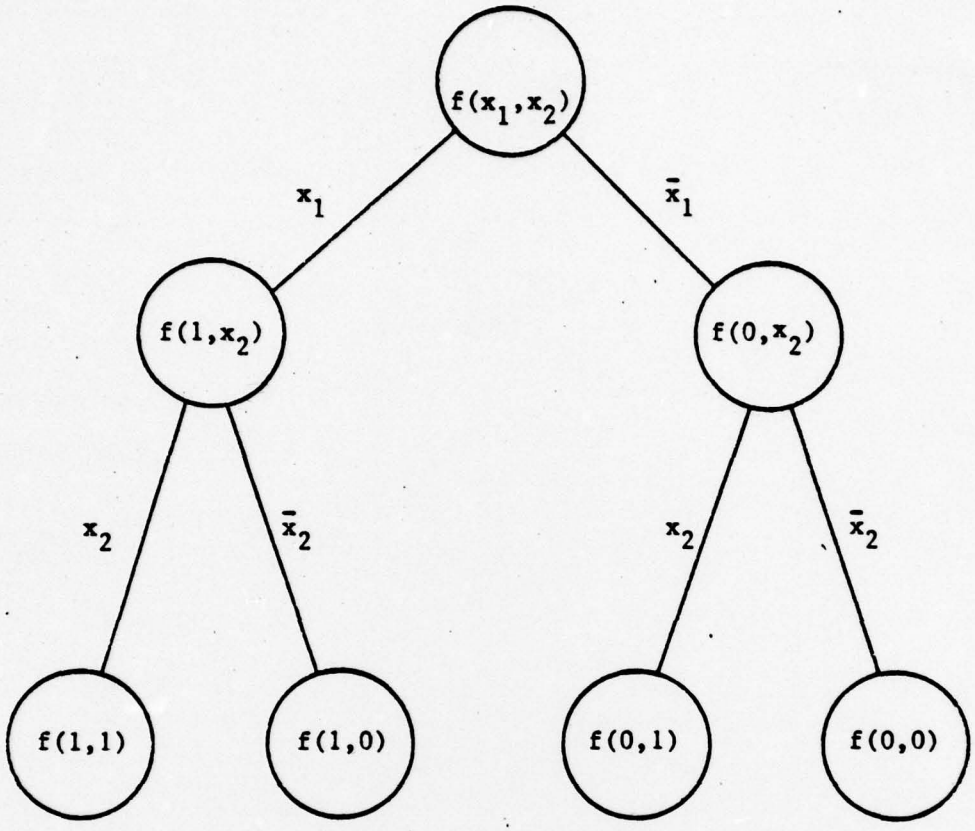


Figure 2-10 Binary Tree Representation of $f(x_1, x_2)$

Each of the leaves of the tree represent the binary coefficients for the chain of Boolean variables $(x_i x_j x_k ..)$ between that leaf and the root of the tree. e.g. $f(x_1, x_2) = x_1 x_2 f(1,1) + x_1 \bar{x}_2 f(1,0) + \bar{x}_1 x_2 f(0,1) + \bar{x}_1 \bar{x}_2 f(0,0)$. Furthermore, at any level of the tree, the Boolean product of the chain of Boolean variables between that node and the root of the tree with the Boolean function represented by the node defines the expansion of the function to that point - e.g., $f(x_1, x_2) = x_1 f(1, x_2) +$

$\overline{x}_1 f(0,x_2)$.

For an incompatibility matrix, I, of a subset of a set of jobs, K, we say that the reduced incompatibility matrix $I_{(j_i)}^{(1)}$ is formed from I by eliminating the row and column associated with j_i of I. Likewise, the reduced matrix $I_{(j_i)}^{(2)}$, is formed by eliminating every row and every column of the incompatibility matrix I that contains a 1 in the column associated with j_i, and by eliminating the row and column corresponding to j_i. In words, $I_{(j_i)}^{(2)}$ is formed from I by eliminating the rows and columns of jobs that are incompatible with j_i and the row and column associated with j_i.

For example,

The matrix I is:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

$$I_{(j_3)}^{(1)} = \begin{array}{c} \\ 1 \\ 2 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

where $I_1(j_3)$ is the reduced incompatibility matrix formed by eliminating the row and column associated with job j_3

and

$$I_2(j_3) = \begin{matrix} \\ 1 \end{matrix} \begin{matrix} 1 \\ \left[\ 0\ \right] \end{matrix}$$

where $I_2(j_3)$ is the reduced matrix formed by eliminating every row and column of I that contains a 1 in the column associated with job j_3 and the row and column associated with j_3.

Note that I_1 and I_2 in turn are incompatibility matrices of a subset of the original set, and can themselves be reduced in the same manner.

Also, we say that $u(j_i)$ is the string of jobs from the incompatibility matrix I that are incompatible with j_i. (i.e. there is a one in the column corresponding to j_i in each of the rows of I that is associated with each job in the string.) For example, $u(j_3)$ above is equal to $j_2 j_4$ for the incompatability matrix I above.

A Boolean funtion, f, as represented above , is true when there are no longer any incompatible elements that remain about which it can

be expanded. In terms of the incompatibility matrix, once the matrix becomes equal to zero or becomes null, its Boolean function representation is true. $I_2(j_3)$ above is, for example, true as would be the case for any matrix in which all of the rows and columns had been eliminated. It is at this point that further expansion of the Boolean function is not necessary.

From the definitions above, it is easy to see that the matrix I_2 is in fact a submatrix of the matrix I_1 since I_2 can be formed from I_1 if every column and row "pointed to" by the column corresponding to j_i in I is eliminated from I_1.

Yang demonstrated that given an incompatibility graph and the corresponding Boolean function $f(j_i,....j_n)$, the expansion of f using Shannon's first expansion with respect to a variable j_k, and the reduction of the corresponding incompatibility graph with respect to the same variable are equivalent and there is a one-to-one correspondence between the Boolean products of $f(j_1,j_2,..,j_{k-1},1,j_{k+1},...j_n)$ and the elements of $I_1(j_k)$. Also, there is a one-to-one correspondence between the Boolean products of $f(j_i,..j_{k-1},0,j_{k+1},..j_n)$ and the elements of $u(j_k)I_2(j_k)$.

As mentioned previously, the determination of the sum of products representation of the Boolean function $f(j_1,...,j_n)$ can be

represented by a binary tree with weighted edges. Hence, in order to determine the sum of products form of the Boolean function f, we must traverse the binary tree equivalent of f and determine the elements of each node of the tree as well as the edge weights as defined by the Shannon Expansion of the function. To traverse the tree structure, we will make use of a preorder traversal.

A preorder traversal of a tree involves the following sequence of events:

Process Node

Proceed to Left Successor

Proceed to Right Successor

We will make use of an algorithm given by Berztiss (Ber) for preorder traversal of a binary tree, but with two modifications. First, the exact structure of the nodes of the tree is not known a-priori, but will be generated as the tree is traversed; and, in addition to creating a push down stack for the nodes of the tree we will also create a stack of the edge weights of the tree and the reduced incompatibility matrix associated with each node.

The edge weights for the tree refer to the job strings that are part of each Boolean product term. For ease of reference, we will refer to the edge weights as j_i or $\bar{j}_i$, where $\bar{j}_i$ represents the string of jobs $u(j_i)$.

There are several properties of the trees we will be examining that will be used in the generation and traversal of each tree.

Since I_2 for any incompatibility matrix I (either reduced or the original) is zero or null if I_1 is zero or null, no right branch from any node of the tree will exist if there is no left branch from that node. In addition, because the tree will branch at a node only if the count of ones in some column of the incompatibility matrix associated with that node is greater than zero and any one in a column must be in a row with a different index than the column, if there exists a left successor to a node, there will also be a right successor. Thus, each node of the tree has a successor if and only if it has both a left and a right successor.

The algorithm below generates all of the maximal compatibles for a given job set from the incompatibility matrix of that job set.

Algorithm 2.1

Step 1

Given an incompatibility matrix I, find the maximum number of ones in any column. If the maximum number is zero go to Step 3.

Step 2

(a) Push the job index j_i of the lowest indexed column with the maximum number of ones onto the "Job" stack.

(b) Push $\bar{j}_i$ onto the "Job Complement" stack.

$$(\bar{j}_i = u(j_i))$$

(c) Push $I_1(j_i)$ on to the "Array" stack.

(d) Set $I = I(j_{l_i})$

(e) Go to Step 1.

Step 3

(a) Store complement of "Job" stack with respect to job set as an irredundant maximal compatible.

(b) If "Job" stack empty, stop.

Else

(c) Pop "Job" stack

(d) Pop "Job Complement" stack.

(e) Push $\bar{j}_i$ onto "Job" stack.

(f) Store complement of "Job" stack with respect to job set as an irredundant maximal compatible.

(g) Pop "Job" stack twice. If stack becomes empty, stop.

Else

(h) Pop "Job Complement" stack.

(i) Push $\bar{j}_k$ onto "Job" stack.

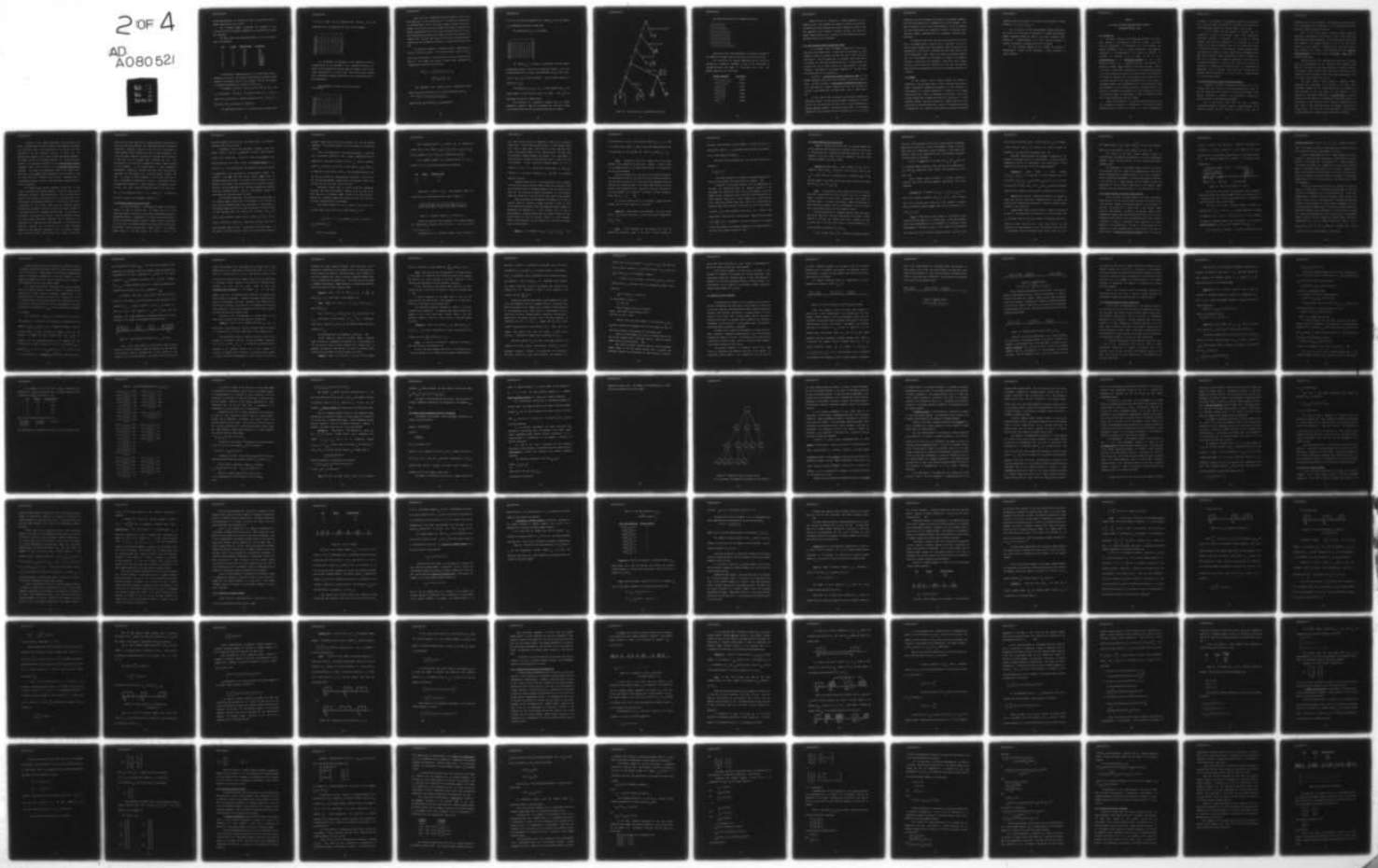
(j) Pop "Array" stack twice, and push $I(j_{2_k})$ onto "Array" stack.

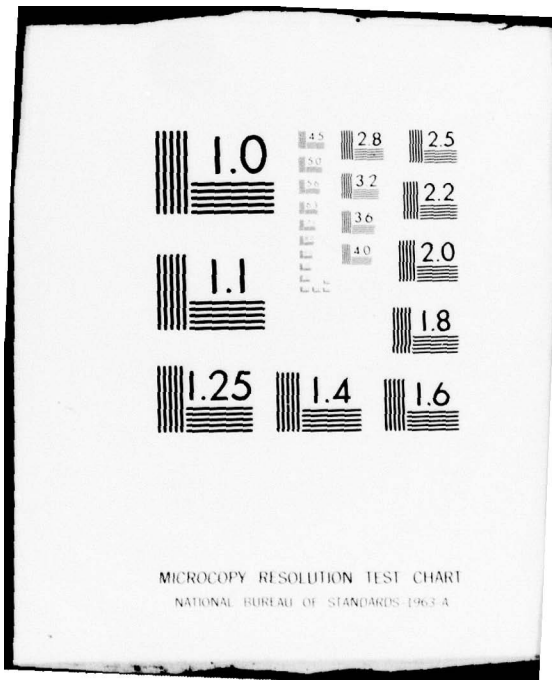
(k) Set $I=I(j_{2_k})$ Go to Step 1.

The result of applying the above algorithm to the

AD-A080 521 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
OPTIMAL MULTIPROCESSOR SCHEDULING OF PERIODIC TASKS IN A REAL-T--ETC(U)
DEC 79 W D SEWARD
UNCLASSIFIED AFIT/DS/EE/79-2 NL

2 OF 4
AD A080521




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

incompatibility matrix of a given set of jobs is the collection of all maximal compatibles of that job set.

The following example illustrates the formation of the collection of all of the maximal compatibles for the given job set using this algorithm.

The table below lists the characteristics of a set of periodic jobs to be scheduled.

Job	Period	Execution Time	Load Factor
1	1	0.5	0.5
2	1	0.4	0.4
3	2	1.6	0.8
4	2	1.4	0.7
5	3	0.5	0.1667
6	4	0.5	0.125
7	6	0.8	0.1333
8	8	1.5	0.1875
9	8	0.6	0.075
10	9	0.5	0.0556

We successively examine each pair of the tasks listed above to determine if the sum of the execution times exceeds the greatest common divisor of the periods of each of the job pairs. That is, we determine the pairwise compatibility of the jobs in the job set.

For example, $E_1 + E_2 = 0.9 \leq \gcd(T_1, T_2) = 1$ such that the jobs j_1 and j_2 are compatible. The sum of the execution times for j_1 and j_3 is equal to 2.1 units while the greatest common divisor of T_1 and T_3 is 1. The jobs j_1 and j_3 are clearly not compatible.

The compatibility matrix for the entire job set is shown below.

A one in a column i and row k signifies that the jobs j_i and j_k are compatible; and a zero signifies that they are not compatible.

	1	2	3	4	5	6	7	8	9	10
1	1	1	0	0	1	1	0	0	0	1
2	1	1	0	0	1	1	0	0	1	1
3	0	0	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	0	0	1	0
5	1	1	0	0	1	1	1	0	0	1
6	1	1	0	1	1	1	1	1	1	1
7	0	0	0	0	1	1	1	0	1	1
8	0	0	0	0	0	1	0	1	1	0
9	0	1	0	1	0	1	1	1	1	0
10	1	1	0	0	1	1	1	0	0	1

We can represent the complement of the compatibility relation, incompatibility, be the complement of the compatibility matrix or a graph of the incompatibility relation among the elements of the job set. Either representation can in fact be used, only the particular technique used to determine the maximal compatibles of the set determines which is more advantageous.

The incompatibility matrix for the job set above is as follows:

	1	2	3	4	5	6	7	8	9	10
1	0	0	1	1	0	0	1	1	1	0
2	0	0	1	1	0	0	1	1	0	0
3	1	1	0	1	1	1	1	1	1	1
4	1	1	1	0	1	0	1	1	0	1
5	0	0	1	1	0	0	0	1	1	0
6	0	0	1	0	0	0	0	0	0	0
7	1	1	1	1	0	0	0	1	0	0
8	1	1	1	1	1	0	1	0	0	1
9	1	0	1	0	1	0	0	0	0	1
10	0	0	1	1	0	0	0	1	1	0

Each one in the incompatibility matrix signifies that the two jobs represented by the row and column of the matrix are incompatible. As presented above the row and column numbers coincide with the job numbers in the job set; but once the algorithm is begun to determine the maximal compatibles for the job set, and the original matrix is reduced, the correspondence between the job number and the row or column will no longer exist. The rows and columns will have the job they represent indicated by the job number at the head of a column and the start of the row.

We begin the proceedure by finding the lowest number job with the most ones in its column. For the matrix above, column 3 (job j_3) has 9 ones in its column. It will therefore be the first variable about which we will expand the Boolean function that represents the incompatibility relations of the job set.

$$f(j_1, j_2, \ldots\ldots, j_n) = j_3 f(j_1, j_2, 1, j_4, \ldots j_n) + \bar{j}_3 f(j_1, j_2, 0, j_4, \ldots j_n)$$

The complement of the variable j_3 can be represented by those jobs in the job set that have a one in the row associated with j_3,

i.e., $u(j_3) = j_1, j_2, j_4, j_5, j_6, j_7, j_8, j_9, j_{10}$.

These are the jobs with which j_3 is incompatibile.

All of the jobs from the original job set except j_3; but, only because j_3 is incompatible with each of these jobs.

The reduced matrix $I(j_1 j_3)$ is therefore,

	1	2	4	5	6	7	8	9	10
1	0	0	1	0	0	1	1	1	0
2	0	0	1	0	0	1	1	0	0
4	1	1	0	1	0	1	1	0	1
5	0	0	1	0	0	0	1	1	0
6	0	0	0	0	0	0	0	0	0
7	1	1	1	0	0	0	1	0	0
8	1	1	1	1	0	1	0	0	1
9	1	0	0	1	0	0	0	0	1
10	0	0	1	0	0	0	1	1	0

The matrix $I(j_2 j_3)$ is formed by eliminating from the original incompatibility matrix every row and column that contains a one in the column associated with j_3 along with eliminating the j_3 row and column. In this case, $I(j_2 j_3)$ is the null matrix. Thus no further reduction is possible for $I(j_2 j_3)$.

The function $f(j_1, j_2, 1, j_4, \ldots j_n)$ is next expanded about j_4, the lowest indexed job with the most ones in its column. (Job j_8 has the same number of ones but a higher index.)

The functions are successively expanded until no further expansion is possible. Figure 2-11 represents the tree that is formed as the Boolean functions and the associated matrices are reduced.

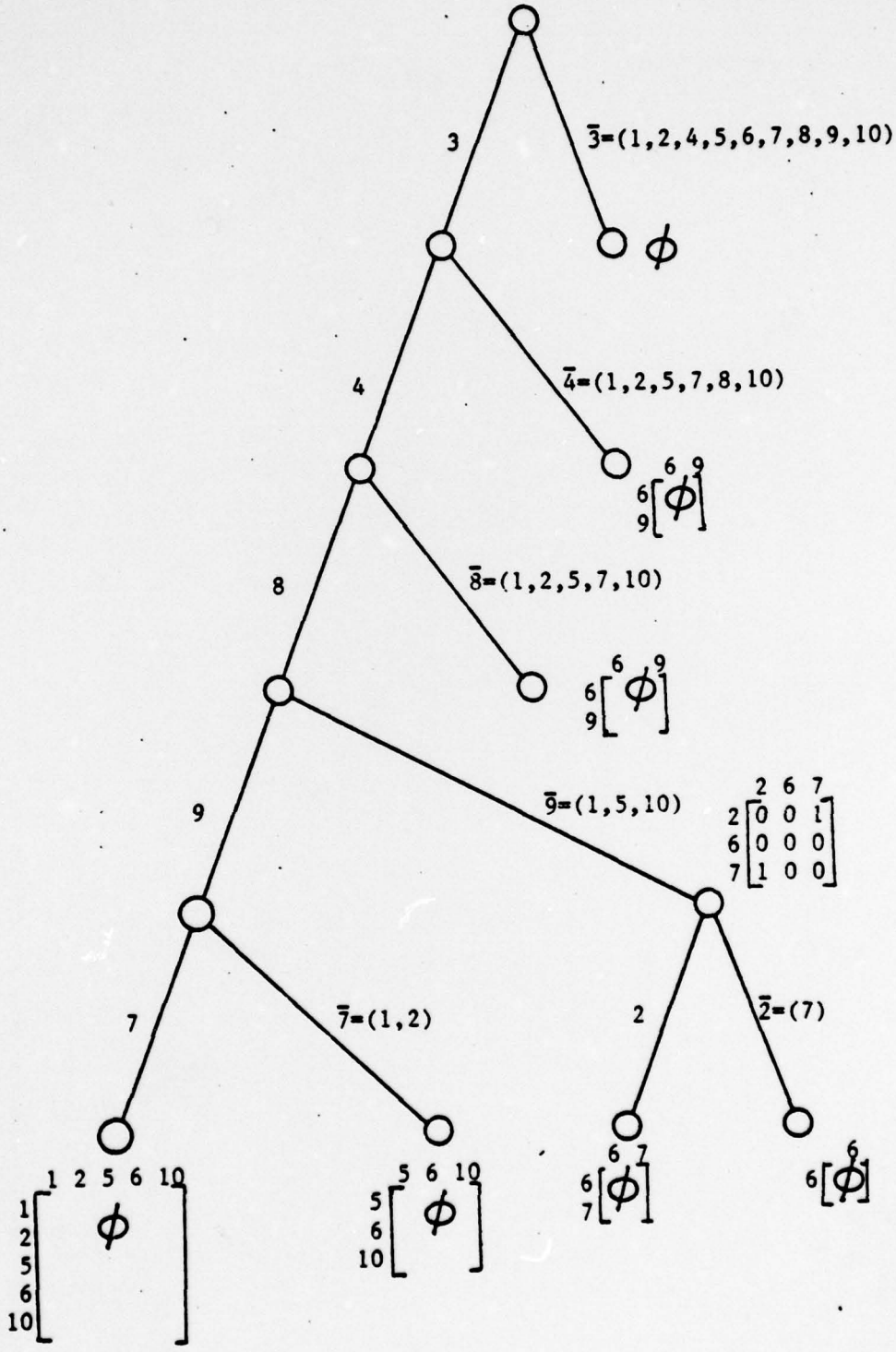


Figure 2-11 Tree Structure for Incompatibility Function

The resulting collections of incompatible jobs are:

```
(j ,j ,j ,j ,j )
  3  4  7  8  9
(j ,j ,j ,j ,j ,j )
  1  2  3  4  8  9
(j ,j ,j ,j ,j ,j ,j  )
  1  2  3  4  5  8  10
(j ,j ,j ,j ,j ,j ,j  )
  1  3  4  5  7  8  10
(j ,j ,j ,j ,j ,j ,j  )
  1  2  3  4  5  7  10
(j ,j ,j ,j ,j ,j ,j  )
  1  2  3  5  7  8  10
(j ,j ,j ,j ,j ,j ,j ,j ,j  )
  1  2  4  5  6  7  8  9  10
```

These sets of jobs were determined by collecting the strings of job numbers of the edges from each leaf node to the root of the tree.

The collection of all maximal compatibles for this job set is determined by finding the complement of each of the collections of incompatibles above with respect to the original set of jobs. The maximal compatibles are listed below.

Maximal Compatible	Load Factor
A=(j ,j ,j ,j ,j) 1 2 5 6 10	1.2473
B=(j ,j ,j ,j) 5 6 7 10	0.4806
C=(j ,j ,j) 6 7 9	0.3333
D=(j ,j ,j) 2 6 9	0.6000
E=(j ,j ,j) 6 8 9	0.3875
F=(j ,j ,j) 4 6 9	0.9000
G=(j) 3	0.8000

These then are the collection of maximal compatibles of the example job set. They represent the largest collections of jobs that do not exclude each other pairwise; and as such, represent the sets of jobs that may have a valid schedule on a single processor. Of course, the maximal compatibles are not all disjoint, and any schedule will have to partition the job set by some means.

2.6 Load Consistent Maximal Compatibles (LCMC)

In the example above, the maximal compatible "A" has a total load factor that exceeds unity. As was pointed out previously, no valid schedule can exist on a single processor for a set of jobs if the total load factor of the jobs exceeds one. So, although none of the jobs in "A" excludes the other when considered pairwise, no valid schedule can be found with all of these jobs on the same processor. We must, therefore, find subsets of the maximal compatible "A" that have total load factors that do not exceed unity.

We define a Load Consistent Maximal Compatible (LCMC) as any maximal compatible which has a total load factor less than or equal to unity. Further, we define a Load Consistent Compatibility Class, LCC, to be any subset of a maximal compatible for which the load factor of all jobs in the subset does not exceed unity. Note, a LCMC is also a LCC.

At this time, we will not address any particular technique for efficiently finding the load consistent maximal compatibles of a given job set. We will only point out that it can be done by examining each maximal compatible formed by the algorithm given above. If the maximal compatible is load consistent then we do nothing; but, if it is not load

consistent, then we will examine every subset of that maximal compatible and retain each subset that is irredundant and is load consistent. This type of technique could in fact be very laborious, but at this time we are only concerned with the existence of a method of finding the load consistent maximal compatibles and not necessarily concerned about efficiency.

In passing, we also note that the total load factor for the job set in the example above is equal to 3.1431. Thus there is no set of three or fewer processors that could possibly fulfill the execution time requirements of the job set. Therefore, the total load factor of any job set establishes a lower bound on the number of processors required; but, in a later chapter, we will expand on bounds for the number of processors required and will establish a lower bound that is tighter than that given by the total load factor of a job set. The new lower bound that we will establish will make use of the maximal compatibles of a job set.

2.7 Summary

In this chapter, we have formally defined the problem of scheduling a set of periodic tasks. We have shown that there exists a simple technique for examining the feasibility of the existence of a valid schedule on a single processor for a subset of the original collection of jobs. First, we determined for every pair of jobs in the job set whether they excluded each other from existing in a single processor schedule; and, then we formed the largest collections of the jobs that did not exclude each other. Finally, we addressed the problem of limiting the total execution time resources required for any maximal

compatible collection of jobs to that which can be provided by a single processor, i.e. $E_i/T_i + \ldots E_n/T_n \leq 1$.

But, we have not at this time determined whether there exists for a given set of jobs that are contained in a single load consistent maximal compatible (except for LCMC of two or less elements) a valid schedule on a single processor; or, if more than one processor is required, the number of processors which will be required.

In the following chapter, we will address the problem of determining for a given LCC the existence of a valid schedule on a single processor.

CHAPTER 3

AN OPTIMAL ALGORITHM FOR UNIPROCESSOR SCHEDULING OF INDEPENDENT PERIODIC TASKS

3.1 Introduction

The primary objective of any scheduling algorithm is to find a valid schedule, if one exists,for a given set of tasks. It is only after the establishment of the existence of a valid schedule that further refinements, such as loading restrictions, interconnection constraints, or reconfigurability of the system are possible. An algorithm for scheduling a set of tasks is said to be an optimal algorithm or an optimization algorithm if, for a given particular instance of a scheduling problem, the algorithm will determine an optimal valid schedule, if one exists (GGJ,Gon). This is in contrast to an efficient algorithm that is not optimal which may not find a valid schedule for a given set of tasks, when in fact one exists. Or an efficient algorithm that will determine a valid schedule which is not optimal relative to the given performance metric. In the case of a performance measure that is dependent only on the determination of any valid schedule, as is our present problem, an optimal algorithm is one which will determine a valid schedule, any valid schedule, for a given set of periodic tasks on a uniprocessor. It will not determine from the set of valid schedules any particular element.

Based on the definition of an optimal algorithm given above, we are interested in being able to determine if a valid single processor schedule exists for a given set of tasks. For, as will be demonstrated

in Chapter 4, the algorithm for determining a minimal set of processors for a job set is based on the ability to determine the existence of a valid uniprocessor schedule for any given subset of the job set.

In this chapter, we will develop an algorithm that will determine for a given set of periodic tasks a valid nonpreemptive uniprocessor schedule, if one exists. We will begin by defining necessary and sufficient conditions for the existence of a valid schedule of any pair of compatible jobs. We will extend these results to arbitrary collections of load consistent pairwise compatible jobs, and formulate the problem as a mixed integer linear programming problem. Next, we will develop a theory of equivalence classes of schedules, and make use of the concept in the implicit enumeration of the collection of all possible schedules for a given job set. Finally, we will define an algorithm for scheduling a set of periodic tasks on a uniprocessor and demonstrate that it is optimal in the sense that the algorithm will determine some valid schedule for the set of jobs on a single processor if a valid schedule exists.

3.2 Previous Results for Single Processor Scheduling

In Chapter 1, we briefly discussed the previous work of Liu and Layland, Serlin, and Labetoulle on multiprogramming periodic tasks. We will now relate the work of the above authors and uniprocessor (single machine) scheduling in general to the nonpreemptive scheduling of periodic tasks in a hard-real-time environment.

All of the previous work which was done to determine uniprocessor schedules of periodic tasks has allowed preemption of the tasks and required only that each active interval of each task be

completed prior to a given deadline. The deadline is usually set equal to the next request for the same task, i.e., the deadline for each task equals the repetition period of that task.

Each of the above authors developed dynamic scheduling algorithms which evaluated the priorities of each active task during system execution using the deadlines of the tasks for which service had been requested. The highest priority of all of the task active intervals yet to be completed is assigned to the task active interval whose deadline is the nearest. These algorithms are called deadline-driven scheduling by Liu and Layland and relative urgency (RU) by Labetoulle and Serlin.

The RU algorithm, which reevaluates the priorities of the tasks which have been requested but not completed at each instance of time based on the proximity of each task's deadline, was shown by Labtoulle to be optimal. That is optimal in the sense that it will create a valid schedule for any set of tasks if there is any other algorithm which will also create a valid schedule for that set.

The other authors listed previously, developed dynamic algorithms which reevaluated the priorities of the tasks, whose computation requests for the current frame had not been completed, at each occurrence of each new request for task execution or each completion of some task's computation. The current frame of a task is defined to be that interval of time between the most recent request for the execution of that task and the next request for computation of the task. Each of these algorithms allows the processor utilization to approach unity, that is the sum of the load factors of the jobs which could be scheduled on a single processor approaches the maximum.

In addition to the dynamic algorithms, Liu and Layland, and Serlin developed fixed priority scheduling algorithms which determine job priorities based on the frequencies of the tasks, and these priorities do not vary during the execution of the job set. The priorities are assigned to the tasks in order of decreasing frequency - the highest priority to the highest frequency job. The fixed priority scheme of Liu and Layland is refered to as the rate monotonic priority assignment, while Serlin called it the intelligent fixed priority algorithm. Each scheme was independently shown to be optimal in the sense that no other fixed priority assignment rule could schedule a task set which could not be scheduled using a priority assignment based on nondecreasing frequency of tasks. These algorithms, as mentioned above, are both preemptive.

The general single machine scheduling problem that is most closely associated with defining schedules for periodic tasks involves scheduling jobs against specified deadlines. While the penalty which results when a deadline is exceeded may vary, successful results have been obtained using weighted tardiness as a criteria when there exists no constraint on the job start times (Rin); or combining weighted tardiness with a weighted earliness criteria which assigns a penalty if a job starts prior to its targeted start time (Sid). In the context of our problem, this implies that the only acceptable schedules are those for which no tasks exceed their deadlines. But, in addition, to the requirement that tasks not exceed their deadlines, our problem has the additional constraint that no task may fail to begin processing immediately when requested; furthermore, there is no specific targeted start time for the first active interval of any jobs, but each active

period after the first must start an integral multiple of the job period after the first active interval. The algorithms previously developed by others have, in general, made use of the fact that there exists an optimal schedule without inserted idle time or without preemption. Only in the case where a penalty is incurred for earliness does the insertion of idle time into the schedule help to minimize the cost. This unfortunately is not the case for scheduling periodic tasks as defined in our environment. As will become obvious in later development, nonpreemptive schedules of periodic tasks which have a hard-real-time requirement for task requests will in general require inserted idle time. It is easy to state an example of a set of jobs which fail to have a nonpreemptive schedule which is valid on a uniprocessor yet can be preemptively scheduled on a single processor. For example, the pair of jobs j_1 and j_2 with periods and execution times of $T_1=2$, $T_2=5$, $E_1=1$, and $E_2=1$ are incompatible since $E_1 + E_2 > \gcd(T_1, T_2)$. Yet they have been shown to have a valid preemptive schedule (LL).

3.3 Accessible Regions for Periodic Tasks

In the previous chapter, we developed the concept of restricted regions for pairs of periodic jobs. We showed that all of the restricted regions for any pair of jobs could be determined by solving a Diophantine equation. Furthermore, if the sum of the execution times of the two jobs did not exceed the greatest common divisor of the periods of the two jobs, the restricted regions of one job relative to the other are disjoint, and are periodic with a period equal to the greatest common divisor of the jobs' periods. The width of each of the

restricted regions for any job pair was found to equal the sum of the execution times of the two jobs.

In actuality, we are more interested in regions in the time domain in which a job, say j_k, may begin an active interval and not conflict with another job j_i. This is of course the complement of the restricted regions and will be called the accessible regions of j_k relative to j_i. The characteristics that defined the restricted regions are complemented in the definition of the accesssible regions. For instance, a job must have every active interval begin within an accessible region with respect to the other job of each pair for a valid schedule to be possible; and, the boundaries of the accessible regions, which are also the boundaries of the restricted regions, are contained within the accessible regions. That is, the accessible regions are closed while the restricted regions are open.

The accessible regions are also periodic with a period equal to the greatest common divisor (gcd) of the periods (T) of the pair of jobs, and each accessible region has a width equal to the difference of the greatest common divisor of the jobs' periods and the sum of the pair of jobs' execution time's (E). That is, for two jobs j_i and j_k, the width of each accessible region is equal to $\gcd(T_i,T_k)-(E_i+E_k)$; and the period of repetition is $\gcd(T_i,T_k)$. When $\gcd(T_i,T_k)=E_i+E_k$, the width of each accessible region is zero - a single point in the time domain. In all cases, if the execution time sum for a job pair does not exceed the

greatest common divisor of the jobs' periods, (i.e., the two jobs are compatible) the accessible regions of one job relative to another are disjoint.

From the construction of the restricted regions, we also know that an accessible region for a job j_k begins immediately after the completion of each active interval of the job j_i. Hence an accessible region of j_k relative to j_i will have a lower bound at $t_{0i}+E_i$, where t_{0i} is defined as the start time of job j_i. The accessible regions for any job pair can therefore be determined from the periods of the two jobs and the resulting greatest common divisor, the execution times of the two jobs, and the start time of one of the jobs.

Henceforth, we will have no need to solve the Diophantine equations that define the accessible regions for any job pair, but we will make use of the greatest common divisor of the jobs' periods to generate the accessible regions from a given job start time for one of the jobs as demonstrated by the following:

The accessible regions of a job j_k relative to job j_i have a set of lower bounds, $A_l(i,k)$, and a set of upper bounds, $A_u(i,k)$.

$$A_l(i,k)=\{t > 0 : t=t_{0i}+E_i+m\gcd(T_i,T_k)\} \text{ and } A_u(i,k)=\{t > 0 : t=t_{0i}-E_k+(m+1)\gcd(T_i,T_k)\}$$

where m is any integer.

Each accessible region of j_k relative to j_i is defined by an ordered pair of real numbers $\langle a_l(m_0), a_u(m_0)\rangle$ where $a_l(m_0)$ and $a_u(m_0)$ are the elements of $A_l(i,k)$ and $A_u(i,k)$ respectively for $m=m_0$.

As an example, consider the accessible regions of a job j_2 relative to a job j_1 with the characteristics shown below.

Job	Period	Execution Time
j_1	3	0.2
j_2	5	0.5

Assume that j_1 starts at $t_{01}=2$. The accessible regions are illustrated by the noncrosshatched areas shown in Figure 3-1.

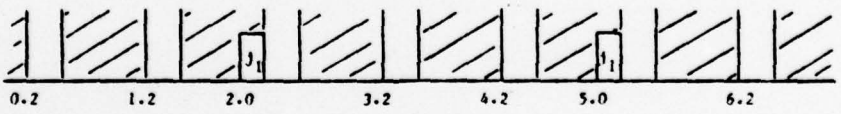


Figure 3-1 Accessible regions of j_2 relative to j_1

The sets of upper and lower boundaries of the accessible regions are respectively $A_u(1,2)=\{0.5,1.5,2.5,3.5,4.5,....\}$ and $A_l(1,2)=\{0.2, 1.2,2.2,3.2,4.2,....\}$.

The generation of the accessible regions of any job relative to

any other job with which it is compatible is seen to be a relatively simple task. We have shown previously that if a valid schedule exists for a pair of tasks, every active interval of each task must start within an accessible region of that task relative to the other task. The accessible regions determine the interval in the time domain in which the active intervals of each task must occur if there is to be a valid schedule. In fact, since the accessible regions of a job j_k relative to a job j_i are periodic with period equal to $\gcd(T_i, T_k)$, either all of the active intervals of j_k will start in accessible regions or none will.

Although we know that for a compatible pair of jobs, a schedule will be valid as long as the start time of every active interval of each job is contained within an accessible region of that job relative to the other job, we as yet have no means of determining if a valid schedule exists for a given set of jobs and the job start times of that schedule.

Before discussing the determination of schedules for any pair of jobs and extend the results to larger collections of jobs, we will further restrict the regions in the time domain that must be examined in determining the existence of a valid schedule. This is done by restricting the accessible regions that can possibly contain the first active interval of each job and still result in a valid schedule. First, we will show that any valid schedule of a collection of periodic jobs is itself periodic.

Lemma 3.1 If a schedule $J=\{j_1(t_{01}), j_2(t_{02}), \ldots j_n(t_{0n})\}$ is a

valid schedule for a given set of periodic tasks $K=\{j_1, j_2, \ldots j_n\}$, then J is periodic with a period T_J equal to the least common multiple (lcm) of the periods of all of the jobs in K. That is, $T_J = \text{lcm}(T_1, T_2, \ldots T_n)$.

Proof The period of any valid schedule must be an integer multiple of the periods of the jobs in the schedule. In fact, it must be an integer mulitple of the least common multiple of the periods of the jobs in the schedule.

Since the periods of the jobs may not vary within any schedule, the relationship of each of the job's first active interval relative to the first active interval of the other jobs in the schedule must be the same after any integer multiple of the least common multiple of the job periods. This includes the first occurrence of the lcm. The period of any valid schedule of a given set of periodic jobs must, therefore, equal $\text{lcm}(T_1, T_2, \ldots, T_n)$. QED

We will now restrict the set of accessible regions which may contain the first active interval of a given job.

Lemma 3.2 If there exists a valid schedule J for a set of jobs $K=\{j_1, \ldots, j_n\}$, then for each job j_i in K the start time is bounded above by $t_{0i} \leq T_i - E_i$.

Proof In our hypothesis we have required that each job execute with periodicity equal to its period. We have allowed no

variation at any time within a valid schedule. In order for a job j_i to execute with a period T_i, j_i must execute some time within the interval $[0,T_i]$ of any schedule containing j_i.

Therefore, with an execution time of E_i the job's start time is bounded by

$$0 \leq t_{0i} \leq T_i - E_i .$$

If the above inequality is violated the required periodicity of the job will not satisfy the constraints of our problem. QED

In addition to the obvious results of the two lemmas above, we can make two *observations* which are important relative to the requirement for inserted idle time within a valid schedule and to the number of active intervals of each of the jobs within each period of a valid schedule. First, since a valid schedule has a period and each job's first active interval must start prior to its period minus its execution time, there are within each period, T_J, of an n-job schedule, J, exactly T_J/T_i active intervals of each job j_i in J. Finally, there exists within each period of a valid schedule inserted idle time equal to $T_J(1-\sum_{i=1}^{n} E_i/T_i)$. Consequently, the *total utilization* of a processor by any valid schedule is determined completely by the characteristics of the jobs in the schedule and will not vary.

3.4 Valid Schedules for Periodic Tasks

Lemma 3.2 bounds the job start time to accessible regions that do not exceed the job's period minus its execution time. We are now able to define exactly the necessary and sufficient conditions for a valid schedule of any pair of jobs which are compatible. We state these conditions formally in the next theorem.

Theorem 3.1 There exists a valid schedule for any pair of compatible jobs j_i and j_k if and only if the job start time, t_{0i} and t_{0k}, of each of the jobs is contained in an accessible region of that job with respect to the other, and the start time of each job is bounded above by $t_{0i} \leq T_i - E_i$ and $t_{0k} \leq T_k - E_k$.

Proof If there exists a valid schedule for this pair of jobs, then every active interval of each job must start in an accessible region. Therefore the first active interval must start within an accessible region. As shown in the previous lemma, the individual job start times must be bounded above by the difference of the job's period minus its execution time for any valid schedule to exist.

From our construction of accessible regions we know that unless each active interval of each job begins within an accessible region relative to the other job a conflict will occur. Likewise, if a job j_i's start time is not less than or equal to the the difference $T_i - E_i$ no valid schedule is possible for j_i and j_k.

If the job start time of a job is within an accessible region,

then every active interval of that job will begin within an accessible region and no conflict will occur in the schedule. Further, if the job start times are bounded above by the difference of the period and execution times of the respective job then the periodicity constraint for each job will be satisfied.

In fact, for <u>every</u> pair of job start times t_{0i} and t_{0k} which are in accessible regions relative to each other and are bounded above by $T_i - E_i$ and $T_k - E_k$ respectively, there exists a valid schedule for the job pair. QED

We can further extend the results of the above theorem to arbitrary sets of more than two compatible jobs when the set is load consistent.

<u>Corollary 3.1</u> A schedule $J=\{j_1(t_{01}), j_2(t_{02}), \ldots j_n(t_{0n})\}$ for a given load consistent compatible $K=\{j_1, j_2, \ldots j_n\}$ is a valid schedule only if the job start time of each job in the schedule is contained within an accessible region of <u>every</u> other job in K, and for each j_i in K, $0 \leq t_{0i} \leq T_i - E_i$.

<u>Proof</u> If there exists a valid schedule J, there exists a valid schedule for every subset of n-1 or fewer jobs. In the final analysis, each of these subschedules is in turn dependent on a collection of valid subschedules of job pairs, j_i and j_k. Since there is a valid schedule for every pair of jobs within the original schedule, each job j_i must

start within an accessible region of every other job j_k in the schedule J, and each start time t_{0i} for each job in K must not exceed the difference of its period and its execution time. QED.

The above results establish the necessary conditions for any schedule J of periodic jobs to be a valid schedule. We are now ready to determine for any LCC of jobs whether or not there exists a valid schedule for that collection of jobs on a single processor.

Theorem 3.2 There exists a valid schedule $J=\{j_1(t_{01}),\ldots j_n(t_{0n})\}$ for a given LCC, K, iff there exists a set of job start times $T_s=\{t_{01},t_{02},\ldots t_{0n}\}$ such that $t_{0i} \leq T_i - E_i$ for all j_i in K and for every pair of jobs j_i and j_k in K, t_{0i} and t_{0k} are within accessible regions of their respective jobs relative to the other job of the pair.

Proof From the previous theorem and corollary, if a schedule is valid, then each job start time is bounded above by its period and execution time difference and each must be within an accessible region relative to every other job.

Suppose there exists a set of job start times for a collection of jobs K. Also suppose that for every job j_i in K the job start time t_{0i} is bounded above by $T_i - E_i$ and is within an accessible region of j_i relative to every other job in K. Then, since there is no load constraint on a valid schedule for a LCC, and the start time of each job j_i is contained within an accessible region of every other job j_k in K

and bounded above by $T_i - E_i$, there are $\binom{n}{2}$ two-job valid schedules. Hence, no two jobs conflict, so the schedule J is valid. QED

The results of the above theorem completely define the conditions that are necessary and sufficient for a valid nonpreemptive schedule to exist for any collection of load consistent and pairwise compatible jobs.

The significance of the greatest common divisor of the periods of the two jobs, both with respect to the compatibility of the jobs and toward determining the accessible regions of the tasks, is apparent. Prior to pursuing the development of an optimal algorithm for scheduling a set of jobs on a single machine, we will examine further the significance of the the physical implication of the greatest common divisor of the jobs' periods and relate it to the previous work of Soh.

3.5 Critical Interval for Periodic Task Scheduling

As we have seen, if a valid schedule is to exist, the greatest common divisor of the job periods bounds the sum of the execution times of the two jobs. Further, the accessible regions of one job relative to the other are periodic with a period equal to the greatest common divisor of their periods. In addition, the greatest common divisor determines the longest time interval available for each job to start and complete an active interval when the nonreference job's active interval is defined to start in an accessible region adjacent to an active interval of the reference job. In view of this limitation on the existence of a valid schedule and the previous work of Soh (Soh), we will say that the length of the critical interval, CI(i,k), of two jobs

j_i and j_k is equal to the $gcd(T_i, T_k)$. Figure 3-2 illustrates the occurrence of an active interval of j_k that begins at the latest possible time in the accessible region adjacent to the active interval of j_i. The restricted regions of j_k relative to j_i are shown as cross hatched areas.

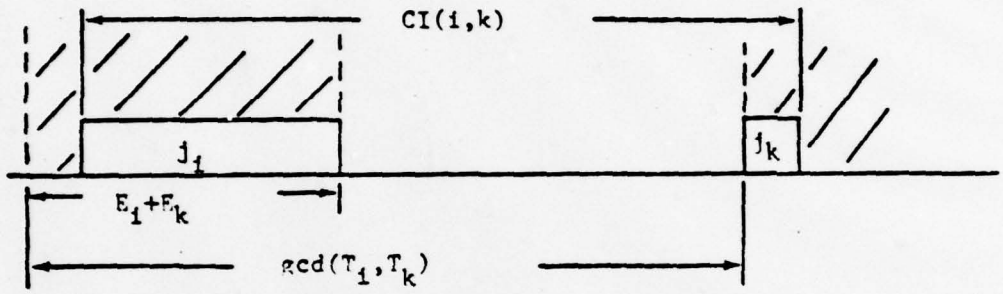


Figure 3-2 The Critical Interval of j_i and j_k

Soh was the first to note the requirement that the execution time sum not exceed the greatest common divisor of the periods, but for a restricted case (Soh). In his dissertation (Soh), Soh defined the critical interval of two jobs as follows:

"Let $J=\{j_k, j_l\}$ be a list of jobs whose schedule is feasible on a processor with $f_k \neq f_l$. That is, j_l is assigned immediately following the completion of j_k. For all u in ITL_k and for all v in ITL_l whenever u-v assumes the minimal positive value, the interval [u,v] is called a Critical Interval of j_k and j_l, and is denoted by $CI(j_k, j_l)$...."

The ITL_k and ITL_l refered to in the above definition are the

initiation time lists, ITL_k and ITL_l, of j_k and j_l respectively and are equal to the respective initiation time sequences (as defined in Section 2.2) with both of the start times assumed to be zero.

Soh then demonstrated that for a compact schedule the critical interval of a job pair equals the greatest common divisor of the jobs' periods. We have shown that the restriction of a compact schedule is not necessary to determine the compatibility of a given pair of jobs by means of the greatest common divisor of the periods. Although we will make no further use of Soh's definition of a critical interval, we will continue to consider the critical interval as equivalent to the greatest common divisor, as it is this time interval that governs the allowable job assignments.

We will now use the previous results for defining the restricted regions of periodic jobs to illustrate the relationship of the active intervals of a nonreference job to a given reference job. We will also examine specific instances in which the greatest common divisor of the periods of more than two jobs defines both a necessary and sufficient condition for the existence of a valid schedule. (Remember, for a pair of jobs the greatest common divisor of the periods defines necessary and sufficient conditions for the existence of a valid schedule.) Although our previous work addressed only two jobs with regard to the greatest common divisor of the job periods, the greatest common divisor of n integers $\gcd(I_1, I_2, \ldots, I_n)$ is defined to equal $\gcd(I_1, \gcd(I_2, \ldots, I_n)) = \gcd(I_m, \gcd(I_k, \ldots, I_r))$ for any I_m, among the n integers. (Ste) We will use this fact in a later development of this section.

Since the period of any valid schedule of a given set of jobs is an integral multiple of the period of each of the jobs, the relationship of all of the jobs in a given schedule can be defined by examining each period of a specified reference job. As we did in Chapter 2, we can examine any schedule of a set of jobs by folding the active intervals of the nonreference jobs into the reference frame.

We showed in the previous development that restricted regions, hence the accessible regions, of any job relative to a reference job are periodic with a period equal to the greatest common divisor of the nonreference job's period and the period of the reference job. We can in a similar manner fold the active intervals of a nonreference job into the reference frame and illustrate the time relationship of the active intervals of the reference job, j_{ref}, to the nonreference job throughout any schedule.

For a given start time t_{0i} of a nonreference job j_i, the job time sequence of j_i is defined by the sequence of ordered pairs of real numbers $<(t_{0i}+mT_i , t_{0i}+mT_i+E_i)>$ for all nonnegative integer m. From the previous chapter, we know that when the elements of the job time sequence, for example the active interval start times $t_{0i}+mT_i$ for integer $m \geq 0$, are considered relative to the reference job, the resulting set of values is defined by the set $\{(t_{0i} \bmod \gcd(T_i,T_{ref}) + k(\gcd(T_i,T_{ref})), k \text{ in } [0, 1,..,(T_{ref}/\gcd(T_i,T_{ref})) -1]\}$. That is, a set of active intervals of a <u>"pseudo" job</u> defined by an execution time E_i

with a period equal to gcd(T_i ,T_{ref}). All of the active intervals of the "pseudo" job are contained within the reference frame and represent the relationship of the active intervals of j_i relative to the active intervals of j_{ref} throughout any schedule of the two jobs. A "pseudo" job $j'_{i,ref}$ is always defined relative to a reference job whose period together with the period of the job j_i determines $T'_{i,ref}$.

For example, given jobs j_1 with E_1 =1 and T_1 =9 and j_2 with E_2 =1 and T_2 =12. If we define j_2 as the reference job, then the period of the "pseudo" job $j'_{1,2}$ is equal to gcd(9,12) = 3 and $E'_{1,2}$ is equal to E_1 . There will exist a valid schedule of $j'_{1,2}$ with j_2 since they are compatible. One of those schedules is illustrated below with the active intervals of the "pseudo" job represented with dashed lines.

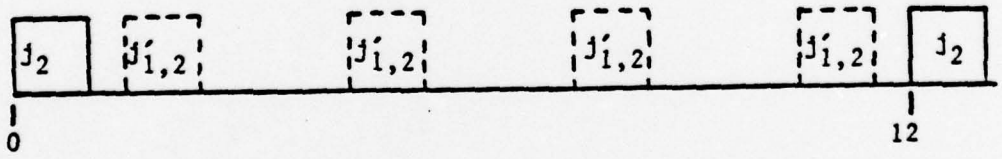


Figure 3-3 Valid Schedule of "Pseudo" Job $j'_{1,2}$ and Job j_2

There are for each "pseudo" job two possible types of active intervals within the reference frame. The "pseudo" jobs may each have active intervals that are coincident with an active interval of the actual job, i.e. the active interval of the actual job has a start time

within the time frame of the first period of the reference job. In the example above, one of those active intervals which start at 1, 4, or 7 must be an active interval of j_1 for a valid schedule to exist. There are also active intervals of the "pseudo" jobs that are only reflections of occurences of active intervals of the actual job in periods of the reference job other than the first. These we will call "reflected" active intervals. If for example, $t_{0i} = 4$ in the above example then all of the remaining active intervals are reflected from other periods of the schedule. Likewise, there may be within the first period of the reference job active intervals of the "pseudo" job for which active intervals of the actual job are coincident with reflected active intervals, or more than one reflected active interval occurs at the same time within the reference frame.

The proof of the following lemma is a direct result of the definitions of "pseudo" jobs and the periods of "pseudo" jobs:

Lemma 3.3 Given a set of periodic jobs $K=\{j_1, j_2, \ldots j_n\}$, if there exists a valid schedule of the set of "pseudo" jobs of K relative to a reference job j_r of K, then there exists a valid schedule for K.

From the previous chapter, we know that there will exist a valid schedule of the "pseudo" jobs only if they are pairwise compatible, i.e., $E_i + E_k \leq \gcd(T'_{i,r}, T'_{k,r})$. It is therefore conceptually easy to determine if the necessary condition for a valid schedule of a collection of "pseudo" jobs is fulfilled. In the general case, however, it is not particularly advantageous to test a given job set for the

existence of a valid schedule of "pseudo" jobs of that set. This is because the compatibility of the "pseudo" jobs is not sufficient for a valid schedule and, except in restricted cases, a valid schedule of a job set is possible when there is a conflict among the "pseudo" jobs. In the following we will examine such a restricted case and illustrate an example of the greatest common divisor as a necessary and sufficient bound on the sum of the jobs' execution times.

Lemma 3.4 Given a job set $K=\{j_1, j_2, .., j_n\}$, if $\sum_{i=1}^{n} E_i \leq$ $\gcd(T_1, T_2, T_3, ...T_n)$ there exists a valid schedule for K.

Proof Suppose n=3, then $E_1 + E_2 + E_3 \leq \gcd(T_1, T_2, T_3) =$ $\gcd(T_1, \gcd(T_2, T_3))$.

Now if we let $T'_{2,3} = \gcd(T_2, T_3)$ and $E'_{2,3} = E_2 + E_3$ define a job $j'_{2,3}$, there exists a valid schedule for $j'_{2,3}$ and j_1. Furthermore, every active interval of j_2 and j_3 can be contained within some active interval of $j'_{2,3}$.

The reasoning above can be extended to arbitrary n. QED

We will employ the results of the above lemma to illustrate cases in which it is necessary and sufficient for a valid schedule of the job set that there be a valid schedule of the "pseudo" jobs. In addition, the execution time sum must not exceed the greatest common divisor of the job periods if a valid schedule is to exist.

Lemma 3.5 Given a job set $K=\{j_1, .., j_n\}$ with each job having

period T, there exists a valid schedule iff $\sum_{i=1}^{n} E_i \leq \gcd(T_1,..T_n) = T$.

Proof Since every job has the same period, the schedule period is also equal to T. There will occur within each period of the schedule one and only one active interval of each job. Therefore if the execution time sum exceeds T no valid schedule is possible.

On the other hand, by the previous lemma there will be a valid schedule as long as the sum of the execution times is not greater than T. QED

A direct consequence of the lemma above is that for the particular job set there will be a valid schedule of K if and only if there is a valid schedule of the "pseudo" jobs.

The sufficient conditions expressed in Lemma 3.4 is for most instances overly restrictive. It seemingly tends toward "overkill" and appears to be of little value except in cases similar to job sets defined in Lemma 3.5. This is, in fact not true, as the next theorem will show.

Theorem 3.3 Given a job set $K=\{j_1,..,j_n\}$ where $\gcd(T_i,T_q) = I$ for all j_i, j_q in K and no two periods are equal, there exists a valid schedule for K iff $\sum_{i=1}^{n} E_i \leq \gcd(T_1,T_2,...,T_n)$.

Proof If the sum of the execution times does not exceed I, then there exists a valid schedule of K.

On the other hand, suppose there exists a valid schedule of K. Since the periods are all different and the greatest common divisor of

each pair is equal to I, the period of an arbitrary job j_i of K can be represented by $T_i = p_i I$ where p_i is a positive integer. The integers p_i and p_r of the periods T_i and T_r respectively must be relatively prime by our hypothesis. That is, $\gcd(p_i, p_r) = 1$. Regardless of the reference job chosen, the periods of each of the "pseudo" jobs relative to the reference job is I. Therefore if there is a valid schedule of the "pseudo" jobs then $\sum_{i=1}^{n} E_i \leq I$.

Suppose that although there exists a valid schedule for K, there is no valid schedule of any set of "pseudo" jobs of K. There must occur within any schedule of the "pseudo" jobs a conflict of active intervals of two nonreference jobs. Since there is a valid schedule of K, at least one of the active intervals must be "reflected". Without loss of generality we will assume that only one of the active intervals is "reflected", say that of a job j_q. Assume also that this active interval conflicts with an active interval of j_i. (Note that if the "pseudo" jobs conflict once, they will conflict at every active interval.) Further we will assume that the reference job is j_r.

The active interval of j_i will occur in the same relation to the reference job at each point in time defined by $m(\text{lcm}(T_i, T_r))$ for all nonnegative integer m. Likewise, if we assume that the occurrence of the active interval of j_q that when "reflected" back resulted in a

conflict with the active interval of j_i occurs at $l_0 T_r$, then there will occur an active interval of j_q in the same relation to j_r at each time $l_0 T_r + n\ lcm(T_r, T_q)$ for all nonnegative integer n.

If there is to be no conflict of the jobs j_i and j_q at the same relative point to j_r then there must be no nonnegative integers n and m such that

$$l_0 T_r + n\ lcm(T_r, T_q) = m\ lcm(T_r, T_i)$$

but this equation is equal to

$$l_0 + np_q = mp_i .$$

Now if integers A and B are relatively prime, there exists natural numbers n and m such that $nA - mB = 1$ (Sie).

Therefore there are natural numbers n and m such that $l_0 + np_q = mp_i$ which contradicts the assumption that a valid schedule can exist for this job set without a valid schedule for the "pseudo" jobs.

There is a valid schedule for a job set as defined above iff the sum of the execution times of the jobs does not exceed the greatest common divisor of the job periods. QED

Thus there are certain restricted cases in which the greatest common divisor of the job periods defines both a necessary and sufficient condition for the existence of a valid schedule of a job set.

We will make use of this fact, in a later chapter, in determining if a given set or subset of jobs has a valid schedule.

In the previous example in this section, we alluded to the existence of schedules of the "pseudo" job that were equivalent. This relationship among valid schedules applies to more than schedules of "pseudo" jobs as we will informally discuss in the next section. In a later section of this chapter, we will develop a formal equivalence relation among schedules of a job set.

3.6 Families of Valid Schedules

We demonstrated in Theorem 3.1 of a previous section that for any pair of compatible jobs there exists a valid schedule for every pair of job start times that are bounded above by their respective period and execution time difference and within an accessible region of the job relative to the other. This of course implies that for any pair of compatible jobs there are a possibly infinite number of valid schedules. Furthermore, as the collection of jobs becomes larger, the problem of specifying possible schedules is compounded.

We begin in this section the development of concepts which will make it possible to enumerate representations of all of the possible schedules for a given set of jobs. In particular, we will show that for an arbitrary collection of jobs, there is a finite set which represents every possible schedule for that job set.

Any processor schedule J is defined by the job start times (which are nonnegative real numbers) of the jobs in that schedule. Of the possibly infinite number of valid schedules for a job set, many are

not really "different" schedules in the sense of job active period sequences, point of reference, and processor idle intervals. They are only different in terms of the real numbers which define the job start times relative to each other.

For example, illustrated below is a single period of a valid schedule for a given set of jobs $\{j_1, j_2, j_3\}$.

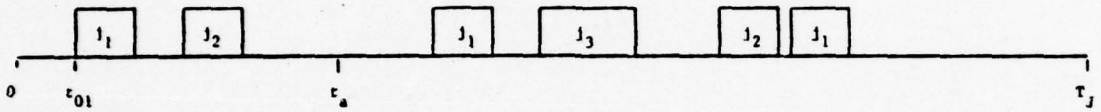


Figure 3-4 The First Period of a Valid Three-job Schedule

Since this schedule is valid and every valid schedule is periodic with a well defined period, the reference point for this schedule, which is shown as t=0 in the illustration above, could, for that matter, be any point within the schedule period which is not wholly contained within any job active period. For example, the reference point could be set equal to t_{01} as shown in Figure 3-5. Or the reference point could be set equal to t_a and the job start times adjusted so that they correspond to different reference time. Figure 3-6 illustrates the schedule with its reference point at t_a of the original schedule's period and its schedule is $J=\{j_1(t_{01}+T_1 - t_a), j_2(t_{02}+T_2 - t_a), j_3(t_{03})\}$. All of these schedules could be considered to be different, but regardless of which of these schedules is examined,

each of the other schedules is "contained" within that schedule. In short, given a set of jobs and a valid schedule for these jobs, there exists a valid schedule for every time point within a schedule's period which is either the start of some job's active interval or within some idle interval of the schedule period.

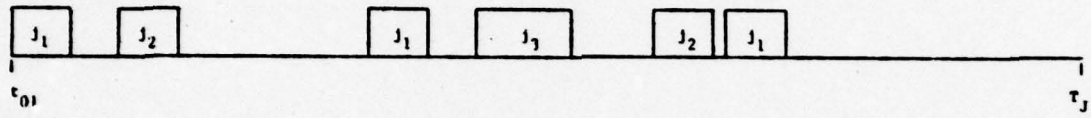


Figure 3-5 Schedule Period for $\{j_1(0), j_2(t_{02} - t_{01}), j_3(t_{03} - t_{01})\}$

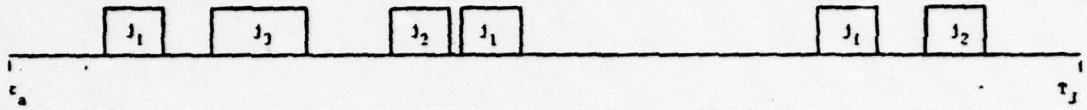


Figure 3-6 Schedule Period
for
$\{j_1(t_{01}+T-t_a), j_2(t_{02}+T-t_a), j_3(t_{03}-t_a)\}$

In the previous illustrations, we did not alter in any way the relative relationships of any of the jobs' active periods to those of the other jobs within the schedule. Suppose that we leave the sequence of job active intervals unchanged but do change the relative occurrance of the job start times as shown below. We see that this too represents a valid schedule for the set of jobs and also contains each of the other schedules defined above.

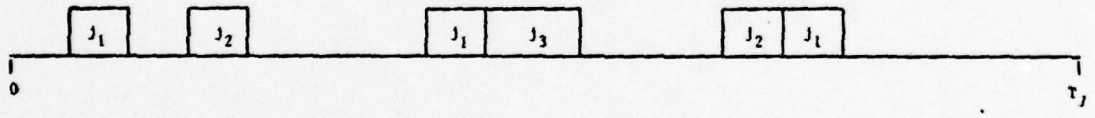


Figure 3-7 Schedule Period for Shifted Jobs j_2 and j_3

While all of the above concepts apply only to valid schedules, we can see that if a possible ordering of job active intervals is to represent a valid schedule these traits must apply. We will say that a <u>family of schedules</u> for a given set of jobs is the set of all possible schedules that can be represented by a given sequence of job active intervals. This is, needless to say, a very loose definition and for

this reason we intend for the concept of families of schedules to be an informal concept. But, we will, in a later section of this chapter, more precisely define this concept and the specific elements of a family of a given collection of jobs. In addition, we will show that for any given job set we can enumerate the representations of all possible schedules by enumerating the families of schedules and defining equivalent families.

We will now proceed to develop the necessary formalism and definitions that we will need to define an optimal algorithm for nonpreemptive scheduling of periodic tasks on a single processor.

3.7 Accessible Regions as Linear Inequalities

To this point, our representation of the accessible regions of one job relative to the other job of a given pair consisted of points in the time domain which defined the upper and lower bounds of each region. This particular representation does not lend itself to mathematical manipulation of job schedules. But each accessible region of one job relative to an other represents an acceptable difference in the job start times of the two jobs, where the accessible regions bound the relative difference of those times. To facilitate the determination of valid schedules for a given job set, we will make use of the formulation of the accessible regions of a job pair as linear inequalities.

To simplify the description of the accessible regions for a pair of jobs, we will arbitrarily choose the job with the higher lexicographic ordered pair of frequency and execution time as the reference job of the pair and define the accessible regions for the other job of the pair by linear inequalities of the difference of the two job's start times.

Assume the job j_i has the highest lexicographic ordered pair of frequency and execution time relative to j_k. The linear inequalities that represent the accessible regions of j_k relative to j_i are constructed as shown in the following lemma:

Lemma 3.6 The set of job start times for a job j_k that are contained in accessible regions of j_k relative to a compatible job j_i, where j_i has a higher lexicographic ordering, is given by the set of linear inequalities

$$\{ E_i + mCI(i,k) \leq t_{0k} - t_{0i} \leq (m+1)CI(i,k) - E_k \}$$

where m is an integer in the set

$$[-T_i/CI(i,k), .. 0, 1, .. T_k/CI(i,k) - 1]$$

Proof First, we will assume that $t_{0i} < t_{0k}$. That is, the first active interval of j_k occurs after the first active interval of j_i.

Since each accessible region for this job pair is $CI(i,k) - E_i - E_k$ wide and the first accessible region for j_k to follow j_i begins at $t_{0i} + E_i$, that accessible region is defined by the bounds on the job start time for j_k given below.

$$t_{0i} + E_i \leq t_{0k} \leq t_{0i} + E_i + CI(i,k) - E_i - E_k$$

or

$$t_{0i} + E_i \leq t_{0k} \leq t_{0i} + CI(i,k) - E_k$$

Because the accessible regions are cyclic with a period equal to the critical interval, every job start time for t_{0k} that is greater than t_{0i} is given by the set of time differences

$$\{ t_{0i} + E_i + mCI(i,k) \leq t_{0k} \leq t_{0i} + (m+1)CI(i,k) - E_k \}$$

where m is a nonnegative integer

or for the same m, the set of time differences

$$\{ E_i + mCI(i,k) \leq t_{0k} - t_{0i} \leq (m+1)CI(i,k) - E_k \}$$

Suppose instead that the job start time $t_{0i} > t_{0k}$, then the job start time difference for this case is given by the set

$$\{ E_i + mCI(i,k) \leq t_{0k} - t_{0i} \leq (m+1)CI(i,k) - E_k \}$$

where m is an integer less than zero.

As shown previously, the start times for each of the jobs is bounded above by the difference of its period and its execution time. Therefore, when t_{0i} exceeds t_{0k}, m must be no less than $-T_i/CI(i,k)$; and when t_{0k} exceeds t_{0i}, m must not exceed $T_k/CI(i,k)-1$. The resulting set of job start time differences is given by

$$\{ E_i + mCI(i,k) \leq t_{0k} - t_{0i} \leq (m+1)CI(i,k) - E_k \}$$

where m is an integer in $[-T_i/CI(i,kk),..,T_k/CI(i,k)-1]$. QED

The set of job start time differences defines <u>all</u> of the accessible regions in which j_i and j_k may start relative to one another.

As an example of how the sets of linear inequalities are generated for a given load consistent set of pairwise compatible jobs, consider the following collection of jobs.

Job	Period	Execution Time
j_1	20	1.75
j_2	20	0.50
j_3	25	0.80
j_4	30	1.50

The critical intervals of each job pair are

CI(1,2)=20 CI(2,3)=5 CI(3,4)=5
CI(1,3)=5 CI(2,4)=10
CI(1,4)=10

The resulting sets of inequalities for each job pair the equation above:

Table 3-1 Start Time Differences for $\{j_1, j_2, j_3, j_4\}$

$-18.25 \leq t_{03} - t_{01} \leq -15.8$
$-13.25 \leq t_{03} - t_{01} \leq -10.8$
$-8.25 \leq t_{03} - t_{01} \leq -5.8$
$-3.25 \leq t_{03} - t_{01} \leq -0.8$
$1.75 \leq t_{03} - t_{01} \leq 4.2$
$6.25 \leq t_{03} - t_{01} \leq 9.2$
$11.75 \leq t_{03} - t_{01} \leq 14.2$
$16.75 \leq t_{03} - t_{01} \leq 19.2$
$21.75 \leq t_{03} - t_{01} \leq 24.2$

$-18.25 \leq t_{04} - t_{01} \leq -11.5$
$-8.25 \leq t_{04} - t_{01} \leq -1.5$
$1.75 \leq t_{04} - t_{01} \leq 8.5$
$11.75 \leq t_{04} - t_{01} \leq 18.5$
$21.75 \leq t_{04} - t_{01} \leq 28.5$

$-18.25 \leq t_{02} - t_{01} \leq -0.5$
$1.75 \leq t_{02} - t_{01} \leq 19.5$

$-19.5 \leq t_{03} - t_{02} \leq -15.8$
$-14.5 \leq t_{03} - t_{02} \leq -10.8$
$-9.5 \leq t_{03} - t_{02} \leq -5.8$
$-4.5 \leq t_{03} - t_{02} \leq -0.8$
$0.5 \leq t_{03} - t_{02} \leq 4.2$
$5.5 \leq t_{03} - t_{02} \leq 9.2$
$10.5 \leq t_{03} - t_{02} \leq 14.2$
$15.5 \leq t_{03} - t_{02} \leq 19.2$
$20.5 \leq t_{03} - t_{02} \leq 24.2$

$-19.5 \leq t_{04} - t_{02} \leq -11.5$
$-9.5 \leq t_{04} - t_{02} \leq -1.5$
$0.5 \leq t_{04} - t_{02} \leq 8.5$
$10.5 \leq t_{04} - t_{02} \leq 18.5$
$20.5 \leq t_{04} - t_{02} \leq 28.5$

$-24.2 \leq t_{04} - t_{03} \leq -21.5$
$-19.2 \leq t_{04} - t_{03} \leq -16.5$
$-14.2 \leq t_{04} - t_{03} \leq -11.5$
$-9.2 \leq t_{04} - t_{03} \leq -6.5$
$-4.2 \leq t_{04} - t_{03} \leq -1.5$
$0.8 \leq t_{04} - t_{03} \leq 3.5$

$5.8 \leq t_{04} - t_{03} \leq 8.5$
$10.8 \leq t_{04} - t_{03} \leq 13.5$
$15.8 \leq t_{04} - t_{03} \leq 18.5$
$20.8 \leq t_{04} - t_{03} \leq 23.5$
$25.8 \leq t_{04} - t_{03} \leq 28.5$

So, even for a small job set, there can be a very large number of constraints to be considered; and by virtue of the disjoint property of the constraints for a given pair of jobs, one and only one of the constraints for each pair of jobs can be satisfied at any given time. For this example, there are 44,500 possible combinations of linear constraints over the job set which must be considered if the nonexistence of a valid schedule is to be demonstrated.

Examination of the set of linear constraints for any given pair of jobs (other than those pairs in which there is a single element) reveals that the constraints are periodic with a period equal to the critical interval of the job pair. We can take, for each job pair, the set of linear constraints each of which have different pairs of upper and lower bounds and rewrite these constraints as a linear inequality with constant upper and lower bounds;

$$E_i \leq t_{0k} - t_{0i} - m_{ik}\, CI(i,k) \leq CI(i,k) - E_k$$

The upper and lower bounds are determined by the characteristics of each job pair as is the integer m_{ik} which is defined by

$$-T_i/CI(i,k) \leq m_{ik} \leq T_k/CI(i,k) - 1.$$

In addition, the start time differences specified by the set of linear equalities define the accessible regions of j_k relative to j_i.

We will define a nonnegative integer s_{ik} as follows:

$$s_{ik} = m_{ik} + T_i/CI(i,k) \text{ so that } 0 \leq s_{ik} \leq (T_i + T_k)/CI(i,k) - 1.$$

Each set of linear inequalities for each job pair can be represented by a linear inequality in the following form:

$$E_i - T_i \leq t_{0k} - t_{0i} - s_{ik} \leq CI(i,k) - E_k - T_i .$$

Each integer s_{ik} defined above has associated with it a job start time difference for the job pair j_i and j_k, and therefore, defines an accessible region of job j_k relative to j_i. We will call the integers s_{ik} schedule elements for reasons that will become clear later.

With the accessible regions of each job pair defined by linear inequalities, the problem of finding any valid schedule for a given LCC can be stated as a standard mixed integer linear programming problem, with one exception - there is no objective function to minimize or maximize. We formally state this fact in the next theorem.

Theorem 3.4 There exists a valid schedule for a given LCC, $K\{j_1,\ldots j_n\}$, if and only if there exists a set of nonnegative real numbers $\{t_{01}, t_{02},\ldots,t_{0n}\}$ and a set of nonnegative integers $\{s_{12}, s_{13},\ldots\ldots,s_{(n-1)n}\}$, where there is an integer s_{ik} for every pair of jobs j_i and j_k in the set, and each integer s_{ik} is bounded above by

$$s_{ik} \leq ((T_i + T_k)/CI(i,k)) - 1$$

and the following set of constraints are satisfied:

$$\{\ E_i - T_i \leq t_{0k} - t_{0i} - s_{ik} \leq CI(i,k) - E_k - T_i\ \}$$

for all j_i and j_k in K where k>i.

Proof Note that the upper bounds on each of the schedule

elements, s_{ik}, induce the upper and lower bounds on the job start times t_{0i} and t_{0k} of $t_{0i} \leq T_i - E_i$ and $t_{0k} \leq T_k - E_k$.

The proof of this theorem follows directly from the results of previous lemmas and theorems and the definition of the integers s_{ik}.

QED

3.8 Mixed Integer Programming Solution to Scheduling

The standard mixed integer linear programming problem can be stated as follows (Dan,Gas):

$$\text{maximize} \quad f(\bar{x},\bar{y}) = \bar{c}_1 \bar{x} + \bar{c}_2 \bar{y}$$

subject to

$$\bar{A}_1 \bar{x} + \bar{A}_2 \bar{y} = \bar{b}$$

and $\bar{x} \geq \bar{0}$ integer, $\bar{y} \geq \bar{0}$

where $\bar{c}_1$ is an n_1 element row vector; $\bar{c}_2$ is an n_2 element row vector; $\bar{A}_1$ and $\bar{A}_2$ are m by n_1 and m by n_2 matricies respectively; $\bar{x}$ is an n_1 element column vector of integers; the column vector $\bar{y}$ contains n_2 elements; and $\bar{b}$ is an m element column vector.

The number of constraints is given by the integer m, while the

number of integer variables is n_1 and the number of real variables is n_2. If n_1 is zero, the above problem statement is a standard linear programming problem i.e., there are no integer constraints.

For our particular problem, that of determining schedules for periodic tasks, the integer variables in the vector $\bar{x}$ are the schedule elements s_{ik}, and the real variables in the vector $\bar{y}$ are the job start times t_{0i} for each job in the job set. The vectors $\bar{c}_1$ and $\bar{c}_2$ are at this point undefined.

In our previous development, the linear constraints were expressed as inequalities, while our statement of the mixed integer linear programming problem requires equality constraints. It is a simple procedure to redefine each of the inequality constraints as equality constraints.

For each of the linear constraints that are inequality constraints, we must introduce an additional positive variable called a slack variable to convert the constraint to an equality constraint (Dan,Gas).

The inequality constraints of the form $t_{0k} - t_{0i} \leq B_1$

become $t_{0k} - t_{0i} + v_{ik} = B_1$

and

constraints of the form $B_2 \leq t_{0k} - t_{0i}$

become equality constraints

explained by using a tree. For example, the representation of a three-job set is illustrated in the next figure.

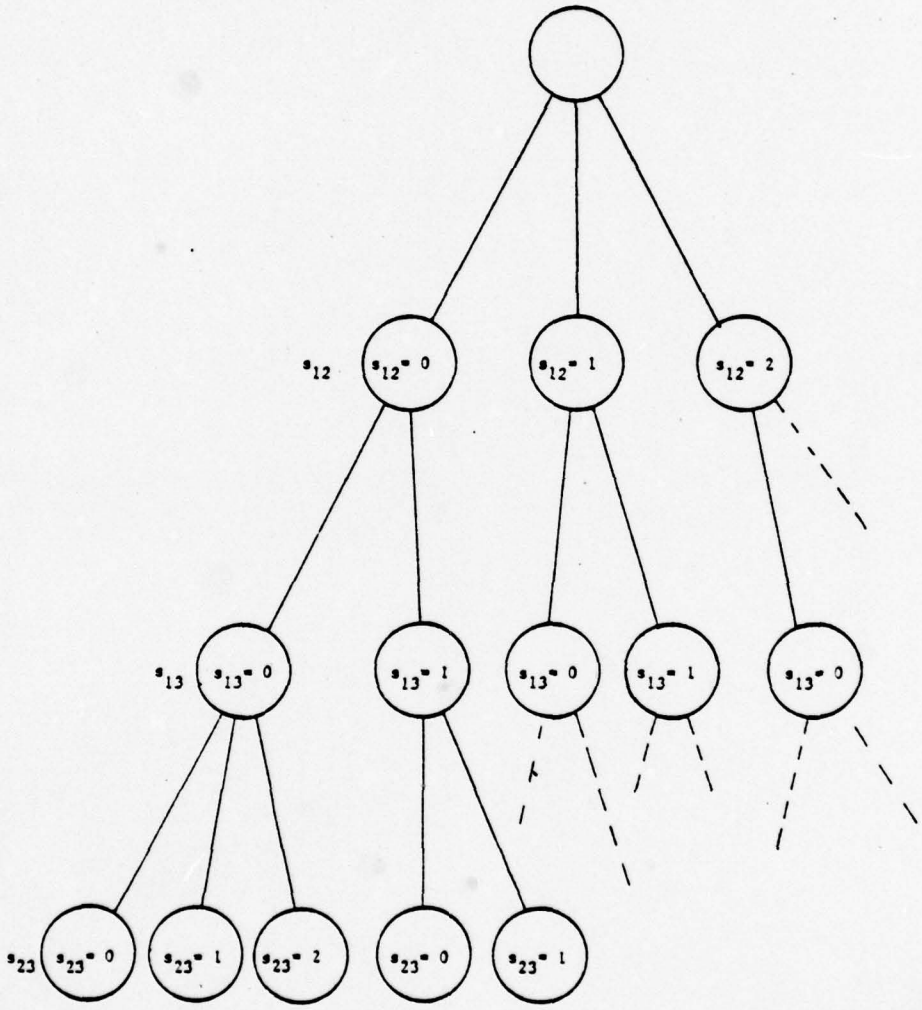


Figure 3-8 Enumeration Tree for a Three-Job Set.

For our purposes, the enumerative procedures for the solution of

the mixed integer problem are based on a branch and bound technique. That is, the procedure branches in its search of the possible solutions by using a performance measure to bound the possible improvement of any solutions yet to be consisdered. Such a procedure may require the exact enumeration of the possible solutions, or employ the characteristics of the particular problem to implicitly enumerate some of the possible solutions.

In the following paragraphs, we will define some of the terminology of mixed integer programming and discuss both the general concepts of linear programming and specific approaches to the solution of our particular problem - the uniprocessor scheduling problem. After introducing the terminology and the basic techniques of solution, we will develop the concepts by which we will attempt to improve the efficiency of our algorithm by reducing the set of possible solutions which must be explicitly examined.

A solution to a given linear programming problem is called <u>feasible</u> if there exists a positive real vector $\bar{y}$ which satisfies the linear constraint $\bar{A}_2 \bar{y} = \bar{b}$. Likewise, a solution to the mixed integer programming problem is called <u>feasible</u> if there exist vectors $\bar{x}$ and $\bar{y}$ of nonnegative integers and real numbers respectively which satisfy the linear constraint equation $\bar{A}_1 \bar{x} + \bar{A}_2 \bar{y} = \bar{b}$. Note that for a solution to be feasible it is not necessary that it maximize the objective function, but the opposite is not true.

We know that any constrained optimization problem can be <u>relaxed</u>

by removing some of the original constraints. For example, the integer requirements for one or more of the integer variables can be eliminated to create a relaxation of the original problem. Also, it is known that if the relaxed problem has no solution, then there is no feasible solution for the problem of which it is a relaxation; and the optimal solution of the original problem is bounded by the optimal solution of any relaxed problems.

A candidate problem is any problem that arises from an attempt to solve the original problem. Any relaxation of that problem or the original problem itself may be considered a candidate problem.

Finally, we say that a candidate problem has been fathomed if it can be determined that further examination of the candidate problem cannot result in a better solution.

The most common approach to solving mixed integer problems by using relaxation is to remove all integral constraints, i.e., allow each variable in the vector $\bar{x}$ to be a nonnegative real number, and solve the resulting linear programming problem. If there is no feasible solution to the relaxed problem, there is no feasible solution to the mixed integer problem and no possible valid uniprocessor schedule can exist for the job set under consideration. If there is a feasible solution to the relaxed problem, the integral constraints of the original problem are added one at a time until either no feasible solution is found or a valid schedule is determined with all of the integer constraints satisfied.

If we refer again to the enumeration tree in Figure 3-8, we see that each node of the tree represents a candidate problem of the

original mixed integer problem. The first node (the root of the tree), for example, represents the candidate problem in which there are no integral constraints. Each successive node in the tree represents a problem which is less and less a relaxation of the original problem. Finally, each of the leaf nodes represents a particular integer vector $\bar{x}$ of the original problem. Because of the representation of the candidate problems as nodes in a tree structure, fathoming a node is equivalent to fathoming a candidate problem.

A natural temptation when a feasible solution is found for any relaxed problem is to pick the nearest integer for any of the variables which do not have integer values. But, it has been shown that the nearest integer may not even be contained in the set of feasible solutions much less represent an optimal solution (Saa).

Our technique of determining the existence of a valid schedule for a given LCC will entail solving a relaxation of the original problem at each node of an enumeration tree. Each level of the tree will have associated with it a schedule element s_{ik} for the given job set. Each node of the tree will have a weight which represents a specific integer value of s_{ik}. The tree is traversed by successively solving candidate problems at each node of the tree and reducing the relaxation of the original problem until all of the possible candidate problems have been considered. Each of the candidate problems is solved as a linear programming problem with a subset of the integer variables constrained to a specific integer value and all other integer variables allowed to assume any positive real value. Thus, our computational procedure to

solve the job scheduling problem requires a procedure for solving the standard linear programming problem as well as, a technique for systematically traversing the tree and solving the mixed integer programming problem.

We have not, to this point, in our development of the single machine job scheduling problem, made any mention of the elements of the linear objective function or, in fact, any particular algorithm for solving the linear programming problem. As we have stated, our objective is the determination of any feasible schedule if one exists. This, as we will see, does not require the use of the objective function per-se. But, in Section 3.14 of this chapter, we will address the problem of determining from the collection of all feasible schedules for a given job set a subset of schedules which are optimal relative to some performance measure. We will express the measure of each schedule's optimality by means of the linear objective function.

The determination of any feasible schedule and the solution of each of the linear programming problems is based on the Two Phase Simplex procedure developed by Dantzig (Dan). The first phase of the Simplex algorithm determines (by forming a linear objective function) an initial feasible solution if one exists, for the set of constraints given. Then, from the initial feasible solution, the second phase of the Simplex algorithm finds in a finite number of steps an optimal solution to the constrained problem for the given objective function. Clearly, if there is no objective function specified, any feasible solution is an "optimal" solution.

Although there are for each job set a finite number of possible

solutions, the actual number of possible solutions may be very large and require an extreme amount of computational resources to explicitly examine each possible solution. For this reason, we will strive to take advantage of the specific nature of the nonpreemptive scheduling problem for periodic jobs to enable us to implicity examine possible solutions. Such an implicit enumeration technique makes use of a performance metric or a particular relationship among possible solutions to fathom nodes; and therefore eliminate some of the possible solutions without explicitly examining each solution below a fathomed node.

For example, as discussed previously, if a valid schedule exists for a given set of jobs, the reference time t=0 for that particular schedule can in fact be assigned to any point in time which is either at the start of some active interval or within an idle interval of the schedule period. In particular, the reference time can be assigned to be the job start time of the lowest indexed job in the LCC or to the start time of any other job in the LCC.

By assigning a job start time of t=0 to the lowest indexed job in the LCC, one real variable, say t_{0i}, is removed from the set of real variables to found. Also, the substitution of $t_{0i}=0$ into the linear inequalities changes all of the linear inequalities relative to t_{0i} to "absolute" bounds on the job start times for each remaining task; and each of the schedule elements s_{ik} associated with the reference job, will be redefined with lower upper bounds such that

$$0 \leq s_{ik} \leq T_k/CI(i,k) - 1.0$$

for s_{ik} as defined above.

Also, each of the linear inequalities which include the reference job j_i are altered to

$$\{\ E_i \leq t_{0k} - s_{ik}\ \ CI(i,k) \leq CI(i,k) - E_k\ \}$$

for s_{ik} as defined above.

The effect of such an assignment of a job start time is the reduction of schedules that must be examined to determine the existence of a valid schedule, and the implication that no job can have an active interval which precedes j_i's first active interval. This assignment of a zero start time to one of the jobs will in no way prevent a valid schedule from being found for a given job set, if one exists; and, no job need be assigned a zero start time for the development of this chapter to work.

In the following sections, we will develop the theory that will enable us to make use of the characteristics of a given set of jobs to determine the existence of a valid schedule for certain relaxations of the original problem without examining each relaxation explicitly. We will also develop techniques for determining for a given valid schedule every schedule whose validity can be implied from the given schedule.

3.9 Vectors of Schedule Elements

In this section we will develop the concepts and techniques necessary to take a given schedule element or collection of schedule elements for a given set of jobs and determine the sequence of the

schedule elements implied by the initial values. We will then use these results to illustrate the creation of a sequence of collections of schedule elements for a specific set of jobs. In later sections we will show how these techniques can be used to form equivalence classes of schedules.

Except when a job is assigned a zero start time, the schedule element for any given job pair defines two things about that job pair's first active intervals. First, the schedule element defines which job of the pair has an active interval which occurs first in the schedule for a given relationship of the jobs' start times. Also, the schedule elements define the relative difference of the jobs' start times. For example, for job j_i and job j_k whose job start times for a valid schedule must be contained in the set $\{ E_i - T_i \leq t_{0k} - t_{0i} - s_{ik} \leq CI(i,k) \leq CI(i,k) - E_k - T_i \}$, $t_{0k} > t_{0i}$ for all $s_{ik} \in [0,1]$ (i.e., t_{0k} occurs prior to t_{0i}), and $t_{0k} > t_{0i}$ for $s_{ik} \in [2,3,4]$ when the first active interval of j_k occurs prior to the first active interval of j_i. For any s_{ik}, the relative difference of the job start times is defined by the upper and lower bounds corresponding to that schedule element.

Each of the possible collections of schedule elements represents a sequence of initial active intervals of the jobs in the LCC being examined. Furthermore, every possible valid schedule for the given LCC of n jobs can be represented by the set of all possible collections of schedule elements. We will say, therefore, that a schedule for a given job set is one of the possible combinations of the schedule elements of

a given LCC, and each collection is some element of some family of schedules.

A schedule for a given load consistent compatible class of n jobs is an $\binom{n}{2}$ vector of nonnegative integers (also called a schedule vector). Each vector element is a schedule element s_{ik} as previously defined where i and k are integers that correspond to job numbers for jobs in the schedule (i.e., for three jobs the schedule vector is $(s_{12}, s_{13}, s_{14}, s_{23}, s_{24}, s_{34})$). The vector elements are ordered by increasing lexicographic ordering of the two element ordered pair (i,k) of job numbers. Each schedule vector, as defined above, does not in general define just a single collection of job start times which represent a valid schedule but a family of schedules as informally defined previously. In fact, each schedule vector defines every schedule (set of job start times) which satisfies the constraints of the mixed integer formulation of the uniprocessor scheduling problem.

The family of schedules introduced in Section 3.6 above suggests that there are schedules, as defined by vectors of schedule elements, that are equivalent, but as yet it has not been shown that there exist equivalence classes of schedules for a given set of jobs. Although finding equivalent schedules by examining a given valid schedule of the original job set would serve no purpose since our goal is to determine the existence of any valid schedule if it exists, finding equivalent schedules for subsets of the job set or equivalent schedules for schedules that are not valid reduces the number of schedules that must be explicitly tested for feasibility.

The following development will show that it is possible to form schedule vectors, which will then be shown to be elements of equivalence classes of schedules, given an initial vector of schedule elements and the characteristics of the jobs in the job set (with a few minor restrictions which will be pointed out).

First it will be shown that it is possible to determine all of the schedule elements that can occur and the order in which they occur given an initial element for any pair of jobs that are compatible. Next it will be demonstrated that schedule element vectors for any three-job subset that contains the LCC reference job can be formed given an initial schedule vector for the set. Before the generation of schedule elements for arbitrary three-job sets is examined, schedule vectors that can never yield a valid schedule are classified and procedures for detecting these impossible "schedules" are demonstrated. This will be done in Section 3.10. Then, in Section 3.11, the schedule element vector sets of three-job sets that do not specify a zero start time for the LCC reference job will be discussed. Finally, the existence and the formation of equivalence classes of schedules will be examined.

Prior to the development of a formulation of equivalence classes of schedules for arbitrary collections of jobs, we will illustrate by example how the schedule elements of a given pair of jobs assumes different values as a function of the equivalent schedules of the job pair is analyzed.

3.9.1 Sequences of Schedule Elements

Shown below is a valid schedule for a pair of jobs j_1 and j_2 with their associated periods and execution times.

Job	Period	Execution Time
j_1	3	0.2
j_2	5	0.5

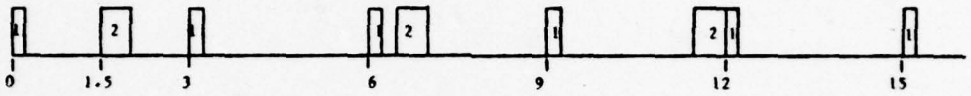


Figure 3-9 Job Pair Schedule

The value of the schedule element s_{12} at the initial active period of job j_1 is determined by the relationship between that active interval and the active interval of job j_2 that starts at 1.5. This is the first active period of j_2 relative to the active interval of j_1 at t=0, and the start time difference of the two active periods determines the job pairs' schedule element. The schedule element s_{12} measured with respect to t_{01}=0 can be determined from Table 3-2 by examination of the accessible region that contains the start time difference $t_{02}-t_{01}$=1.5 and the bounds on the integer s_{12}. It is s_{12}=4.

Also because of the periodic nature of the schedule for these two periodic jobs, relative to the start of the active period of job j_2

at t=1.5, the schedule element s_{12} at t=1.5 is determined by the start of the active interval of job j_1 at t=3.0, the first active interval of j_1 that follows the active interval of j_2 at t=1.5 (Figure 3-9); and the intersection of this start time difference with the bounds of the integer s_{12} (Figure 3-9). It is s_{12}=1. (i.e. $-1.8 \leq t_{02} - t_{01} \leq -1.5$)

In a similar manner, the value of s_{12} at the initiation of each of the active intervals of jobs j_1 and j_2 within the schedule shown can be determined such that there is a sequence of schedule elements $\langle s_{12} \rangle$ for the job pair for this schedule;

$$\langle s_{12} \rangle = \langle 4,1,6,3,0,5,2,7 \rangle .$$

Associated with each element of the sequence is a reference job upon which that schedule element is dependent. There is a one-to one correspondence between the sequence of schedule elements for a given job pair and the associated sequence of reference jobs. The reference job sequence for the above schedule element sequence is:

$$\langle r_{12} \rangle = \langle 1,2,1,1,2,1,2,1 \rangle .$$

Each of the job numbers, which are elements of the reference job sequence, defines the reference job that corresponds to the same element in the schedule sequence. For example, the schedule element 6 is

defined relative to an active interval of j_1 as indicated by the third element of the reference job sequence.

A <u>subsequence of schedule elements</u> is defined as any subset of the original sequence of schedule elements (e.g. <1,3,0,7> is a subsequence of the previous schedule element sequence).

It is clear from the example above that the sequence $<s_{12}>$ contains all possible values of s_{12} for the two jobs being considered (see Table 3-2). This is not a coincedence as Theorem 3.5 demonstrates.

Table 3-2 defines the start time differences of j_2 relative to j_1 and the corresponding schedule element s_{12}. Any start time difference other than those shown represent active intervals that occur outside of accessible regions.

Table 3-2 Start Time Differences $t_{02} - t_{01}$ vs Schedule Elements s_{12}

Start Time Difference	Schedule Element
$-2.8 \leq t_{02} - t_{01} \leq -2.5$	0
$-1.8 \leq t_{02} - t_{01} \leq -1.5$	1
$-0.8 \leq t_{02} - t_{01} \leq -0.5$	2
$0.2 \leq t_{02} - t_{01} \leq 0.5$	3
$1.2 \leq t_{02} - t_{01} \leq 1.5$	4
$2.2 \leq t_{02} - t_{01} \leq 2.5$	5
$3.2 \leq t_{02} - t_{01} \leq 3.5$	6
$4.2 \leq t_{02} - t_{01} \leq 4.5$	7

Theorem 3.5 Every two job sequence of schedule elements $\langle s_{ik} \rangle$, where neither job is the LCC reference job, contains each possible schedule element for that job pair once and only once in every two job schedule of j_i and j_k.

Proof From the original definition of the set of integers s_{ik} that are the schedule elements, the following relations are true:

$$0 \leq s_{ik} \leq (T_i + T_k)/CI(i,k) - 1.$$

$$0 \leq s_{ik} \leq T_i/CI(i,k) + T_k/CI(i,k) - 1.$$

Such that $s_{ik} \in [0,1,\ldots(T_i/CI(i,k) + T_k/CI(i,k) - 1)]$.

The period of the two job schedule, T(i,k), is determined by the least common multiple of the periods of the two jobs as follows:

$$T(i,k) = T_i T_k / \gcd(T_i, T_k)$$

where T(i,k) is the least common multiple of the periods of j_i and j_k.

The number of active intervals of job j_i within the two job schedule is $T_k/CI(i,k)$; and the number of active intervals of job j_k within the schedule is $T_i/CI(i,k)$.

Thus there are exactly as many active intervals of the two jobs within the schedule period as there are elements in the set of schedule elements for that job pair.

It will now be shown that no schedule element can occur more than once within a sequence of schedule elements for a job pair over the two-job schedule period.

Suppose that some element occurs more than once in the sequence for the two-job schedule period. By virtue of the definition of the value of the schedule elements, as related to start time differences within accessible regions, and the periodic nature of the tasks, once the second occurrence of a schedule element occurs the preceding subsequence will repeat - signifying the start of a new two-job schedule period that is less than T(i,k). A contradiction of the schedule period for the two jobs.

Therefore each possible value of schedule element for a job pair must occur once and only once within each two-job schedule period.

QED

The above theorem excludes the sequences generated by a job pair that has the LCC reference job as one of the pair. In those cases, there is no schedule element defined that is relative to the active interval of the nonreference job. But, as will be shown, the same relationship exists for the sequence of schedule elements for a job pair that contains the LCC reference job.

Theorem 3.6 For any load consistent compatible pair of jobs, one of which is the LCC reference job, j_1, the schedule element sequence with respect to the reference job contains every possible schedule element s_{1i} for that pair of jobs over the job pair schedule period.

Proof The number of schedule elements s_{1i} is determined by $T_i/CI(1,i)$ such that s_{1i} is contained in the set

$$[0,1,..,T_i/CI(1,i) - 1]$$

The number of active intervals of j_1 within the two-job processor schedule period is $T_i/CI(1,i)$.

Hence there are as many active intervals of j_1 within the schedule period as there are elements in the set of schedule elements of

the job pair; and again, no schedule element can occur more than once within a sequence of schedule elements for a job pair over the two-job schedule period. QED

Having been assured of the uniqueness of the elements within any pair of jobs' sequence over the two-job schedule period, it is necessary that an algorithm be defined to generate a job pair's sequence given an initial value of the schedule element. This capability is necessary to enable us to determine schedules which are equivalent to a given vector of schedule elements once an equivalence relation is defined.

The periodic nature of a job pair's schedule period guarantees that the sequence of schedule elements will repeat continuously as longer time intervals are considered. So it is only necessary that the schedule element sequence for the schedule period of a job pair be determined once; and then, based on the initial value of the sequence in question the complete sequence can be determined.

Recall the previous example of a two-job set and its resulting schedule element sequence;

Job	Period	Execution Time
1	3	0.2
2	5	0.5

$$\langle s \rangle_{12} = \langle 4,1,6,3,0,5,2,7 \rangle$$

Given the initial element in the sequence, 4, the relationship

of the first active intervals of the jobs, as dictated by the reference job sequence, is defined for a schedule period equal to the least common multiple of the two jobs' periods. The sequence so defined will repeat itself over and over if the time frame of consideration is extended as multiples of the schedule period. A shift of the reference point of the schedule will result in schedule sequences that contain the sequence above but have a different starting element. For example, when the first reference point is chosen as the "first" active interval of j_2 as shown in the illustration above, the sequence of schedule elements will be $\langle s_{12} \rangle = \langle 1,6,3,0,5,2,7,4 \rangle$.

The next theorem demonstrates that the schedule element sequence for a given job pair is unique in the sense that it is cyclic, and given the initial value it determines the entire sequence for that schedule element.

First, we will denote elements in the schedule element sequences by superscripts to indicate the relative positions within the sequences. For example, the initial element in a sequence for a job pair j_i and j_k would be denoted s_{ik}^0; the next element by s_{ik}^1, and so on.

Theorem 3.7 Given two jobs j_i and j_k, $i<k$, $i \neq 1$, and an initial schedule element s_{ik}^0, the schedule element sequence $\langle s_{ik} \rangle$ is determined by the following equation:

$$s_{ik}^{n} = (s_{ik}^{n-1} + T_k/CI(i,k)) \mod[(T_i + T_k)/CI(i,k)].$$

Proof It will first be shown that the succeeding schedule elements, when the initial element is referenced to j_i, are determined by $s_{ik}^{n} = s_{ik}^{n-1} - T_i/CI(i,k)$. Then it will be shown that if the initial schedule element is referenced to j_k, the elements in the sequence are determined by $s_{ik}^{n} = s_{ik}^{n-1} + T_k/CI(i,k)$. Finally, it will be shown that there is a single relationship that exists for generating succeeding elements regardless of the initial reference job.

$T_i/CI(i,k)$ is the number of accessible regions for j_k relative to j_i within each period of j_i. For an initial schedule element that is referenced to job j_i, there are two possible transitions of the reference sequence. Either job j_i repeats as the reference job, or the next reference active period is job j_k. These two cases correspond to no active intervals of j_k within the adjacent idle interval of j_i; and one active period of j_k prior to the next active interval of j_i (no more than one active interval of j_k can occur by virtue of the definitions of the jobs and the relation of their periods) respectively.

For the first case:

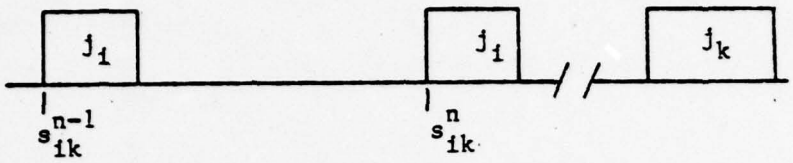


Figure 3-10 Job j_i Repeats as Reference Job

Since $s_{ik}^{n-1} > T_i/CI(i,k)$ (as $s_{ik} > T_i/CI(i,k)$ implies $t_{0i} < t_{0k}$ and j_k does not start in the accessible region adjacent to the active interval defined by the schedule element s_{ik}^{n-1}) and the transition of the reference point from the initial active interval of j_i shown to the active interval of j_i that occurs next in the periodic job's repetition will reduce the number of accessible intervals of j_k relative to the reference job by $T_i/CI(i,k)$, hence for $s_{ik}^{n} > 0$, the value of s_{ik}^{n} is defined as:

$$s_{ik}^{n} = s_{ik}^{n-1} - T_i/CI(i,k).$$

necessary to take a given schedule element or collection of schedule elements for a given set of jobs and determine the sequence of the

For the second case:

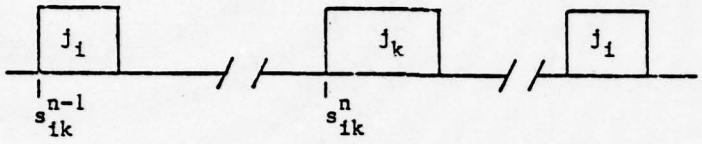


Figure 3-11 One Active Interval of j_k Within the Idle Interval of j_i

the schedule element $s_{ik}^{n-1} \geq T_i/CI(i,k)$ since the next element in the sequence, s_{ik}^{n}, is no longer referenced to j_i, but is referenced to j_k. The next reference job is j_k and its relationship to j_i is determined by the next occurrence of j_i in the time sequence.

Suppose that the active interval of j_k started in the first accessible region to the right of the reference active interval of j_i associated with s_{ik}^{n-1}. By definition then, $s_{ik}^{n-1} = T_i/CI(i,k)$.

It is clear that the value of s_{ik}^{n} =0 would result, as there can be no accessible region for j_k that precedes j_i and is further removed from an active interval of j_i (in this case the second active interval of j_i) and still occurs prior to it.

Hence $s_{ik}^{n} = s_{ik}^{n-1} - T_i/CI(i,k)$

for this particular relationship of j_i and j_k.

Suppose instead that the active interval of j_k was in the second accessible region following the initial reference active interval of j_i.

The value of s_{ik}^{n-1} is clearly $T_i/CI(i,k) + 1$; and the number of accessible regions between the active interval of j_k and the next active interval of j_i is clearly reduced by 1. Thus for this job active interval relationship $s_{ik}^{n} = 1$.

In fact, as the active interval of j_k is shifted to each successive accessible region further to the "right" from the initial reference interval of j_i there is a corresponding reduction of accessible intervals between the active interval of j_k and the next active interval of j_i. Hence s_{ik}^{n} is successively increased by one as defined by:

$$s_{ik}^{n} = s_{ik}^{n-1} - T_i/CI(i,k).$$

Thus for <u>any</u> schedule element sequence that is initially referenced to job j_i, whether the next active interval is j_i or j_k, the next element in the sequence is determined by the above equation.

For an initial schedule element referenced to job j_k, the next element in the sequence must be referenced to job j_i. This is due to the periodic relationship defined for the two jobs. (i.e. $i < k$, then $T_i \leq T_k$)

The value of s_{ik}^{n-1} is limited by:

$$0 \leq s_{ik}^{n-1} \leq T_i/CI(i,k) - 1$$

as these are the only values of s_{ik} for which $t_{0k} < t_{0i}$.

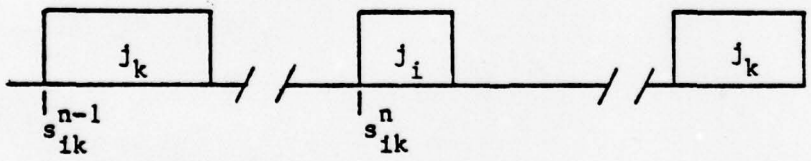


Figure 3-12 Active Interval of j_i Between Active Intervals of j_k

There are $T_k/CI(i,k)$ accessible regions $A(i,k)$ within each period of j_k, and since the two active intervals of j_k are referenced to the same active interval of j_i

$$s_{ik}^{n} = s_{ik}^{n-1} + T_k/CI(i,k).$$

We have developed above two seemingly different relations to determine succeeding elements in a sequence of schedule elements. We will now show that these two relations are in fact equal.

From the definition of the congruence relation, integer A is congruent to integer B modulo m if and only if A-B=km where k is an integer (Ste). Therefore $-T_i/CI(i,k)$ is congruent to $T_k/CI(i,k)$ modulo $(T_i + T_k)/CI(i,k)$ since

$$T_k/CI(i,k) - (-T_i/CI(i,k)) = 1(T_i + T_K)/CI(i,k)$$

and regardless of the initial reference job the next element of the schedule element sequence is defined by

$$s_{ik}^{n} = (s_{ik}^{n-1} + T_k/CI(i,k)) \bmod [(T_i + T_k)/CI(i,k)].$$

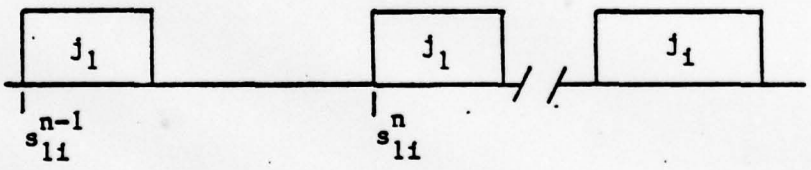
The previous theorem specifically excluded job pairs that contained the LCC reference job. It is known that when one of the jobs is the LCC reference job there are no schedule elements defined that are referenced to any other job. The reference sequence for the schedule elements consists of only the LCC reference job. The method used to generate the schedule element sequences of such job pairs is demonstrated by the following theorem:

Theorem 3.8 For a pair of jobs j_l and j_i, the schedule element sequence is generated from an initial element s_{li}^0 using the equation

$$s_{li}^n = (s_{li}^{n-1} + (T_i - T_l)/CI(l,i)) \bmod [T_i/CI(l,i)].$$

Job j_l is assumed to be the LCC reference job.

Proof $T_i/CI(l,i)$ is the number of accessible regions for j_i within the period T_i. Two possible relationships exist for the active intervals of j_i relative to the active intervals of j_l. Either there is one active interval of j_i within the next idle interval of j_l, or there is no active interval of j_i in the idle interval. Both cases are illustrated below.



Or

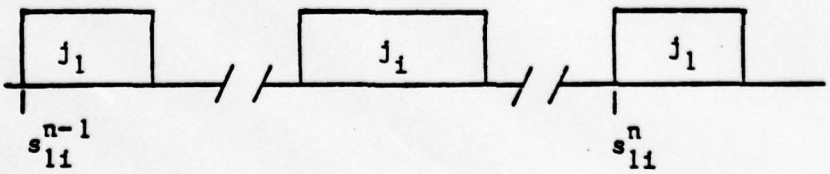


Figure 3-13 Relation of Active Intervals of j_i to j_l

For the case in which there is no active interval of j_i within the next idle interval of j_1, the schedule element is reduced by the number of accessible regions within a period of j_1 so that s_{li}^n is equal to the following:

$$s_{li}^n = s_{li}^{n-1} - T_1/CI(1,i).$$

In the second case, the schedule element is the maximum value of s_{li} minus the number of intervals that precede the next reference interval of j_1. The maximum value of s_{li} is $T_i/CI(1,i)-1$ so that the sequence is defined by

$$s_{li}^n = (T_i/CI(1,i)-1) - (T_1/CI(1,i)-1-s_{li}^{n-1}) \text{ Or}$$

$$s_{li}^n = s_{li}^{n-1} + (T_i - T_1)/CI(1,i).$$

Again because of the congruence relationship, in all cases the schedule sequence is given by

$$s_{li}^n = (s_{li}^{n-1} + (T_i - T_1)/CI(1,i)) \bmod [T_i/CI(1,i)].$$

QED

From the previous theorems, it is clear that the schedule element sequence for any compatible job pair can be generated. This sequence is unique in the sense that it is cyclic; and, given an initial value for the schedule element, the sequence generated is the same as any other sequence generated for that job pair. We will now extend the concept of generating the schedule element sequence to more than two jobs.

In Theorem 3.9 it will be shown that for any three jobs that are contained in an LCC, the schedule element sequences can be determined relative to the reference job of the LCC.

3.9.2 Schedule Vectors with the LCC Reference Job

A previous theorem demonstrated that for any pair of jobs one of which was the LCC reference job the schedule element sequence could be determined by a simple equation. Likewise, it was shown that for a pair of jobs neither of which is the LCC reference job the schedule element sequence for those two jobs relative to each other is also determined by a simple equation. The difficulty that is encountered for a three-job set when the schedule elements are referenced to one of the job's active periods arises in determining, relative to a reference job which is not in the pair, the schedule of the other two jobs relative to each other. Although the two nonreference jobs' schedule element sequence is the same as the two job subsequence for those jobs, relative to the reference job the two-job sequence has not been defined. But, it will be shown that the two-job schedule element sequence relative to the reference job is a subsequence of the two-job sequence when the jobs are considered alone.

For example, in the three job schedule shown below, the sequence of the nonreference jobs' schedule elements is indicated. This schedule element sequence is based on the reference job sequence $\langle r_{23} \rangle$ = <2,3,2,3,2,3,2>.

s_{23} = 3 0 4 1 5 2 6

Figure 3-14. Sequence of j_2 and j_3 Within Three Job Schedule Relative to j_1

It can be seen in this Gantt chart that relative to the reference job the schedule element s_{23} is determined by the nonreference job pair's schedule element immediately to the right (i.e., occurs next in the Gantt chart) of the reference job's active intervals. That is, the schedule element of the first active interval of j_2 or j_3 following the reference job's active interval determines the schedule element s_{23} at the reference job active interval.

The subsequence of $\langle s_{23} \rangle$ that occurs relative to the active intervals of j_1 shown in the above example is:

$\langle s_{23} \rangle$ =<3,4,1,5,2,3>.

It is evident that when referenced to another job, a job pair's schedule element sequence does not consist of each possible schedule element once and only once within the schedule period of the larger set. Even though no apparent pattern seems to exist for generating the schedule element sequence relative to the reference job, it is fortunately relatively easy as the following theorem demonstrates:

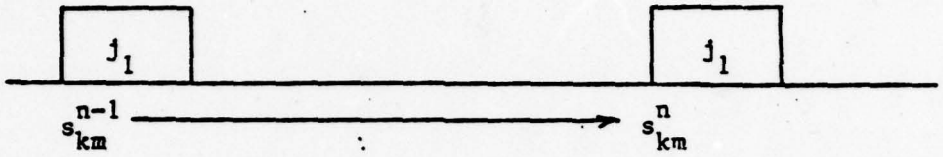
Theorem 3.9 Given the initial values of s_{lk}, s_{lm}, s_{km} the next element in the sequence of s_{km} relative to the LCC reference job, job j_l, is determined by s_{lk} and s_{lm} and the present value of s_{km}, and the two suceeding elements in the two-job sequence for j_k and j_m.

Proof It will first be shown that there are only three possible values for the next element in the sequence of s_{km} relative to the job j_l.

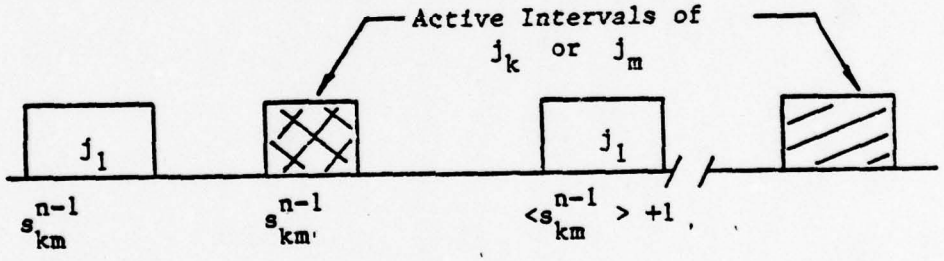
Within any given idle interval of the reference job there can be no more than two active intervals of the other two jobs. This is a result of the definition of the reference job as the job with the highest frequency within the LCC. This makes multiple active intervals of any of the other jobs in the set within one idle interval of j_l impossible.

It is possible for there to be zero, one, or two active intervals of nonreference jobs within an idle interval of j_l. The next element in the sequence relative to j_l is determined as follows:

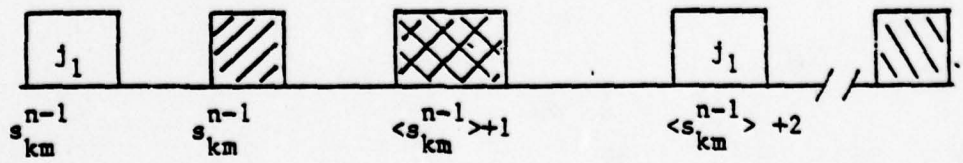
If there are no active intervals of j_k or j_m within the following idle interval of j_l, the value of s_{km} remains the same as the present value.



If there is one active interval of j_k or j_m within the idle interval, the next value of s_{km} relative to j_l is the next element in the original job pair sequence for j_k and j_m.



Finally, if there are two active intervals (one of j_k and one of j_m) within the idle interval, the value of the next element in the sequence $<s_{km}>$ relative to j_l is the second element following the present value s_{km} in the original sequence for the job pair.



It is now necessary that a means be found for determining the number of active intervals of j_k and j_m within the idle interval. The number of active intervals of each of the jobs within the adjacent idle interval of the reference job is determined by the schedule element of j_k and j_m relative to j_l, s_{lk} and s_{lm} respectively.

There are two distinct possibilities for the active intervals of each of the jobs relative to the j_l idle interval:

No active intervals of job j_p, where j_p represents either j_k or j_m for the following discussion, within the idle interval of j_l if and only if

$$T_l/CI(l,p) \leq s_{lp}^{n-1}$$

One active interval of job j_p within the idle interval of j_l if and only if

$$T_l/CI(l,p) > s_{lp}^{n-1}$$

Since the value of s_{lp} is given for each job, j_k or j_m, and the schedule element sequence for each job relative to j_l can be generated

independent of the number of jobs in the set, the schedule element vector for any three-job subset that contains the LCC reference job can be determined relative to j_1. QED

In this section we began the development which will eventually lead to a definition of equivalence classes of schedules. First we developed the techniques required to form the schedule element sequences of any pair of compatible jobs. We then examined the creation of vectors of schedule elements and illustrated the concept of forming schedule element sequences of three-job sets one of which is the LCC reference job. We proved that given an initial schedule element for a job pair j_i and j_k, such that $i<k$ and neither job is assumed to be the specified reference job for the schedule. The schedule element sequence is determined by the equation

$$s_{ik}^{n} = (s_{ik}^{n-1} + T_k / CI(i,k)) \bmod [(T_i + T_k)/CI(i,k)].$$

We also demonstrated that if j_i as defined above were the LCC reference job then the schedule element sequence is defined by

$$s_{ik}^{n} = (s_{ik}^{n-1} + (T_k - T_i)/CI(i,k)) \bmod [T_k / CI(i,k)].$$

Finally we proved that if there is a subset of a given LCC and one of the elements is the LCC reference job j_1, given the initial schedule element vector, each succeeding vector could be formed from the

schedule element sequence of each job pair and the knowledge of the number of active intervals of each nonreference job occurs within the next idle interval of the reference job. There will be no active intervals of nonreference job j_i if and only if $T_1/CI(1,i) \leq s_{1i}^{n-1}$. There will be one active interval of j_i in the next idle interval if and only if $T_1/CI(1,i) > s_{1i}^{n-1}$. The next element for each of the nonreference jobs j_i and j_k relative to the reference job j_1 is defined by the following:

1. If there are no active intervals of j_i or j_k in the following idle interval then $s_{ik}^{n} = s_{ik}^{n-1}$.

2. If there is one active interval of j_i or j_k in the following idle interval then

$$s_{ik}^{n} = (s_{ik}^{n-1} + T_k/CI(i,k)) \bmod [(T_i + T_k)/CI(i,k)].$$

3. If there are two active intervals one each of j_i and j_k in the following idle interval then $s_{ik}^{n} =$

$$(s_{ik}^{n-1} + 2T_k/CI(i,k)) \bmod [(T_i + T_k)/CI(i,k)].$$

In the next section we will examine schedule vectors which can never represent a valid schedule. We will formulate techniques to

detect the occurrence of such vectors, and in Section 3.11 address the generation of schedule elements for sets of jobs larger than two and that do not contain the LCC reference job with a specified start time of zero.

First, let's consider a short example of the formation of schedule vectors of a given set of jobs;

Job	Period	Execution Time
1	2	0.40
2	4	0.40
3	5	0.50

Assume for this example that j_1 is the reference job and every schedule in which we will be interested assigns $t_{01}=0$.

$CI(1,2) = 20$

$CI(1,3) = 10$

$CI(2,3) = 10$

The individual schedule elements of the three jobs are bounded as follows:

$$0 \leq s_{12} \leq T_2/CI(1,2) - 1 = 1$$

$$0 \leq s_{13} \leq T_3/CI(1,3) - 1 = 4$$

$$0 \leq s_{23} \leq (T_2 + T_3)/CI(2,3) - 1 = 8.$$

The schedule element sequences for s_{12}, s_{13}, and s_{23} are determined from the recursive equations listed above.

$$\langle s_{12} \rangle = \langle 0, 1 \rangle$$

$$\langle s_{13} \rangle = \langle 0, 3, 1, 4, 2 \rangle$$

$$\langle s_{23} \rangle = \langle 0, 5, 1, 6, 2, 7, 3, 8, 4 \rangle$$

The sequences above have been formed based only on the characteristics of the job pairs corresponding to each schedule element. There is no relationship among the individual sequences.

Suppose we are given an initial schedule vector

$$\bar{s}_0 = \begin{bmatrix} s_{12} \\ s_{13} \\ s_{23} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 3 \end{bmatrix}$$

and we are to form the sequence of schedule vectors that would result using the recursive equations above and the initial vector.

A <u>schedule vector sequence</u> is the sequence of schedule vectors that is generated from the schedule element sequences of the job pairs given an initial schedule vector.

Each of the elements which are related to j_1 will simply follow their individual sequences. This is due to our definition of sequences relative to a reference job. Schedule element s_{23} will, however, assume values contained in its sequence $\langle s_{23} \rangle$ in a seemingly random fashion, but as defined by the previous equations.

Given the initial vector $\bar{s}_0$, the next value of s_{23} is determined by the number of active intervals of j_2 and j_3 within the idle period of j_1 that follows. There are inequalities listed above which determine the number of active intervals as follows:

$$s_{12}^0 = 0 < T_1/CI(1,2) = 1$$

$$s_{13}^0 = 0 < T_1/CI(1,3) = 2$$

therefore there exists one active interval of each job, j_2 and j_3, in the next idle interval of j_1. The next element $s_{23}^1 = (s_{23}^0 + 2T_3/CI(2,3)) \bmod ((T_2 + T_3)/CI(2,3))$ or $s_{23}^1 = (3+10) \bmod 9 = 4$.

The next vector in the sequence $\bar{s}_1$ is therefore

$$\bar{s}_1 = \begin{bmatrix} s_{12} \\ s_{13} \\ s_{23} \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix}$$

Now $s_{12}^1 \geq 1$ and $s_{13}^1 \geq 2$, so there are no active intervals of j_2 or j_3 in the next idle interval of j_1. Hence there is no change in s_{23}, and the next vector in the sequence is

$$\bar{s}_2 = \begin{bmatrix} 0 \\ 1 \\ 4 \end{bmatrix}$$

This procedure is continued until the vector sequences begin to repeat which they must after $\text{lcm}(T_1, T_2, T_3)/T_1$ vectors. The entire sequence for this example is

$$\langle \bar{s} \rangle = \langle \bar{s}_0, \bar{s}_1, \ldots, \bar{s}_9 \rangle$$

$$\bar{s}_3 = \begin{bmatrix} 1 \\ 4 \\ 5 \end{bmatrix} \quad \bar{s}_4 = \begin{bmatrix} 0 \\ 2 \\ 5 \end{bmatrix}$$

$$\bar{s}_5 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \quad \bar{s}_6 = \begin{bmatrix} 0 \\ 3 \\ 6 \end{bmatrix}$$

$$\bar{s}_7 = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \quad \bar{s}_8 = \begin{bmatrix} 0 \\ 4 \\ 7 \end{bmatrix}$$

$$\underline{s}_9 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \qquad \underline{s}_{10} = \underline{s}_0$$

Before the concept of schedule element sequences is expanded to general three-job subsets of LCC, the relationship of schedule elements to a given schedule will be examined. The schedules will be partitioned into two classes; a class of possibly valid schedules, and a class of schedules that can never be valid.

3.10 Inconsistent Schedule Vectors

As stated previously, any valid schedule of n jobs requires that each subschedule also be a valid schedule. Each subschedule of m jobs is defined by a subset of the schedule vector of the larger set of jobs. Some of the possible combinations of schedule elements for a set of m jobs may represent impossible situations as we will illustrate in the following paragraphs. Such collections of schedule elements need not be considered when a valid schedule is being sought.

A schedule inconsistency results when the elements of a schedule vector or some subvector represents a situation for which it is impossible for a valid schedule to occur. Such a situations are manifest in a contradiction among a collection of schedule elements .

Although inconsistencies occur in any size of schedule vector of three or more jobs, schedule inconsistencies of more than three jobs are a direct result of an inconsistency among at least one of the three-job subsets of the larger job set. For this reason further examination of inconsistent schedules will be directed at inconsistencies in three-job schedules.

Consider a three-job subset of an LCC, $\{j_i, j_k, j_m\}$. None of the jobs in the set are the LCC reference job.

The schedule vector

$$\begin{bmatrix} T_i/CI(i,k) \\ 0 \\ T_k/CI(k,m) \end{bmatrix} = \begin{bmatrix} s_{ik} \\ s_{im} \\ s_{km} \end{bmatrix}$$

for example, is a possible combination of the values of the integers s_{ik}, s_{im}, and s_{km}.

By definition, the first element in the example schedule vector specifies that job j_i precedes j_k; the second element requires that j_m precedes job j_i; the reference sequence implied by these two elements is <m,i,k>. Yet, the last element of the vector implies that job j_k precedes j_m - a clear contradiction. This combination of schedule elements is not consistent and can never represent a valid schedule for the three-job set or any larger set that contains this subvector in its schedule vector.

We will now define inconsistent schedule vectors and how they are manifest. We will then show how they can be detected from the schedule elements of a given job set.

There are two levels of inconsistencies possible for schedule vectors. Both levels are best described as inconsistencies of predecessor/successor relationships as defined by the schedule vector.

The example above is representative of an Level-1 (L1) inconsistency. It is an inconsistency that is evident from examination of the implied reference job sequence for a given schedule vector. An L1 inconsistent schedule can be determined from the schedule elements of the schedule vector.

L1 inconsistencies cannot occur if one of the jobs in the three-job subset is the LCC reference job. As defined previously, the reference job must always precede all other jobs in the schedule (this is reflected in the definition of the schedule elements of all other jobs relative to the LCC reference job); and for a three job set the relationship of the two nonreference jobs to each other is determined completely by the schedule element that is associated with that pair.

The second level of inconsistency, Level-2 (L2), is related to the relative difference of the starting times of the jobs. Inconsistencies of this type result when the accessible regions defined by a given schedule vector's elements contradict each other with respect to the possible start time differences. Such an inconsistency is illustrated below for a reprsentative three job set.

Schedule Elements		Accessible Regions
$s_{23} = 0$	then	$-19.5 \leq t_{03} - t_{02} \leq -1.2$
$s_{24} = 0$	then	$-29.5 \leq t_{04} - t_{02} \leq -13.33$
$s_{34} = 0$	then	$-38.8 \leq t_{04} - t_{03} \leq -33.33$

The accessible regions given above are the regions defined by the schedule elements shown. But, from the schedule elements above job

j_4 must be the first job in the reference sequence. Yet, if $t_{04}=0$, then the start times for j_2 and j_3 must be as follows:

$$33.33 \leq t_{03} \leq 38.8$$

$$13.33 \leq t_{02} \leq 29.5$$

which results in the following relationship of these two jobs' start times:

$$3.83 \leq t_{03} - t_{02} \leq 25.47$$

an impossible situation given the schedule element s_{23} previously defined in the table above.

Note that the above schedule is not L1 inconsistent since the schedule elements imply only one reference job sequence <4,3,2>.

Obviously then, if a schedule is L1 inconsistent then it is L2 inconsistent also; but, a schedule that is L2 inconsistent need not be L1 inconsistent. In fact, if the LCC reference job is contained in the three-job set an L1 inconsistency is not possible. When the LCC reference job is in the set, it is by definition the reference job and the relationship of the other two jobs is defined by their common schedule element.

For any schedule vector that contains any three job subschedule that is inconsistent, there is no valid schedule possible. Further examination of each of these levels of inconsistencies makes it possible

to determine how to detect an inconsistent schedule when it is occurs. First, a procedure for detecting L1 inconsistencies will be defined.

The schedule element for each job pair defines a precedence relation for those two jobs. This precedence relation is reflected in the value of the schedule element, for example s_{ik}, relative to a threshold, $T_i/CI(i,k)$, that defines which job precedes the other in that schedule.

$$s_{ik} \geq T_i/CI(i,k) \text{ implies } j_i \text{ precedes } j_k$$

while

$$s_{ik} < T_i/CI(i,k) \text{ implies } j_k \text{ precedes } j_i .$$

The precedence relation of the two jobs is therefore a binary relation represented by a boolean variable b_{ik} where:

$$b_{ik} = 0 \text{ iff } s_{ik} < T_i/CI(i,k)$$

and

$$b_{ik} = 1 \text{ iff } s_{ik} \geq T_i/CI(i,k).$$

Of the eight possible combinations of the three boolean variables that represent the schedule elements of a given three-job set, two will always be L1 inconsistent. The other six will never be L1 inconsistent.

The two cases that are L1 inconsistent are:

$$\begin{bmatrix} b_{ik} \\ b_{im} \\ b_{km} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

and

$$\begin{bmatrix} b_{ik} \\ b_{im} \\ b_{km} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

The reader may want to verify that these two relations are the only possible L1 inconsistent relations for a three job set.

Translated into schedule elements a three job set is L1 inconsistent for any of the following conditions:

$$s_{ik} < T_i/CI(i,k)$$

and

$$s_{im} \geq T_i/CI(i,m)$$

and

$$s_{km} < T_k/CI(k,m)$$

or

$$s_{ik} \geq T_i/CI(i,k)$$

and

$$s_{im} < T_i/CI(i,m)$$

and

$$s_{km} \geq T_k/CI(k,m).$$

As an example, for

$\{s_{ik}\}=\{0,1,2\}$ where $T_i/CI(i,k)=1$

$\{s_{im}\}=\{0,1,2,3,4\}$ where $T_i/CI(i,m)=2$

$\{s_{km}\}=\{0,1,2,3,4,5,6\}$ where $T_k/CI(k,m)=2$,

any combination of

$$\begin{bmatrix} s_{ik} \\ s_{im} \\ s_{km} \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \text{ or } 3 \text{ or } 4 \\ 0 \text{ or } 1 \end{bmatrix}$$

or

$$\begin{bmatrix} s_{ik} \\ s_{im} \\ s_{km} \end{bmatrix} = \begin{bmatrix} 1 \text{ or } 2 \\ 0 \text{ or } 1 \\ 2 \text{ or } 3 \text{ or } 4 \text{ or } 5 \text{ or } 6 \end{bmatrix}$$

is L1 inconsistent.

Schedules that are L2 inconsistent are not as easily detected. The determination that a schedule is L2 inconsistent requires the examination of the linear constraints for the relative start times of each of the job pairs of the three-job subsets of the job set in question.

For any three jobs j_i, j_k, j_m with schedule element vector $\overline{S}$ and the associated linear constraints

$$A_1 \leq t_{0k} - t_{0i} \leq B_1$$
$$A_2 \leq t_{0m} - t_{0i} \leq B_2$$
$$A_3 \leq t_{0m} - t_{0k} \leq B_3$$

a valid schedule is possible only if

$$A_2 - B_1 \leq B_3$$

and

$$B_2 - A_1 \geq A_3$$

(or any of the equivalent relations of the upper and lower bounds of the start time differences) are satisfied.

In the case shown, the relations represented by the bounds on the linear constraints guarantees that for the schedule elements which define the first two constraints, it is possible for start time differences of j_k and j_m to be within the last linear constraint defined by the last schedule element of the three jobs.

That is, for

$$A_2 - B_1 \leq B_3$$

and

$$B_2 - A_1 \geq A_3$$

then,

$$A_3 \leq B_2 - A_1 \leq t_{0m} - t_{0i} \leq A_2 - B_1 \leq B_3 .$$

These requirements for consistency of the schedule elements make it possible to determine if a given three job subschedule is L2 consistent by testing the validity of the above relative start times as implied by the given schedule elements.

The bounds of the linear constraints can be expanded into the terms that form them. This results in linear constraints for the schedule elements that are a function of the other two schedule elements as shown below:

$$A_2 = E_{im} + s_i - CI(i,m) - T_i$$

$$B_1 = CI(i,k) - E_{ik} + s_k - CI(i,k) - T_i$$

and

$$B_3 = CI(k,m) - E_{km} + s_m - CI(k,m) - T_k$$

such that

$$E_i + s_{im}\,CI(i,m) - CI(i,k) + E_k - s_{ik}\,CI(i,k) \leq CI(k,m) - E_m + s_{km}\,CI(k,m) - T_k$$

or

$$E_i + E_k + E_m + T_k - CI(i,k) - CI(k,m) + s_{im}\,CI(i,m) - s_{ik}\,CI(ik) \leq s_{km}\,CI(k,m)$$

Also

$$B_2 = CI(i,m) - E_m + s_{im}\,CI(i,m) - T_i$$
$$A_1 = E_i + s_{ik}\,CI(i,k) - T_i$$
$$A_3 = E_k + s_{km}\,CI(k,m) - T_k$$

and

$$B_2 - A_1 \geq A_3 \quad \text{then}$$

$$s_{km}\,CI(k,m) \leq T_k - E_k - E_i - E_k + E_m + CI(i,m) + s_{im}\,CI(i,m) - s_{ik}\,CI(i,k)$$

So that for L2 consistency

$$E_i + E_k + E_m + T_k - CI(i,k) - CI(k,m) + s_{im}\,CI(i,m) - s_{ik}\,CI(i,k) \leq s_{km}\,CI(k,m) \leq T_k - E_k - E_i - E_k + E_m + CI(i,m) + s_{im}\,CI(i,m) - s_{ik}\,CI(i,k)$$

Every three-job subset of each larger set of jobs being examined for a valid schedule defines such a linear constraint of one schedule element as a function of the other two schedule elements.

The linear constraints as defined by the constraint above are not limited to integer values. The bounds are in general not integer, and are not guaranteed to be positive. The bounds that will actually be used requires the use of the greatest integer and the least integer

functions and the additional constraint that the schedule elements be within the absolute bounds defined for each element by the accessible regions.

The resulting constraints are:

$$\{(E_i + E_k + E_m + T_k - CI(i,k) - CI(k,m) + s_{im} CI(i,m) - s_{ik} CI(i,k))/CI(k,m)\} \leq s_{km} \leq [(-E_i - E_k - E_m + T_k + CI(i,m) + s_{im} CI(i,m) - s_{ik} CI(i,k))/CI(k,m)]$$

where $[A]$ is the greatest integer $\leq A$, $\{B\}$ is the least integer $\geq B$ and $0 \leq s_{km} \leq (T_k + T_m)/CI(k,m) - 1$.

The significance of the above inequality will become evident when we define the algorithm for constructing a valid schedule. The inequality defines the upper and lower bounds for consistency of a given schedule element in terms of the other two schedule elements of a three-job set.

3.11 Equivalence Classes of Schedules

There are, as shown above, two classes of schedule vectors. One class represents those schedules which may represent a valid schedule. The other class consists of those schedule vectors that are inconsistent. Each of these schedule classes are important for the particular method we will use to find a single processor schedule. If it is possible to form equivalence classes of schedules and to define the elements of each equivalence class, then once a consistent schedule is tested and found to not be a valid schedule, every equivalent schedule will also be known to be not valid. Also, if a schedule is found to be inconsistent, every schedule that contains that particular

combination of schedule elements will also be inconsistent. It must be shown that no inconsistent schedule is contained in the same class as consistent schedules. Otherwise, equivalence classes of schedules, as we will define them, cannot exist.

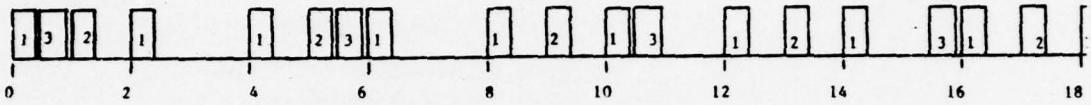
We will now incorporate the techniques developed in Section 3.9 for generating schedule element sequences and sequences of schedule vectors and extend these results to arbitrary subsets of an LCC.

From the previous development, we know that each schedule element sequence is cyclic and only appears to be non-cyclic when the element sequence is referenced to a job not represented by that particular element. Yet, the individual elements will repeat each integral multiple of the least common multiple of the associated jobs' periods. Thus, regardless of the reference job for any given three-job set, each schedule vector will repeat at a time equal to the period of the three-job processor schedule period.

There are further ramifications of this property. Since the schedule vector sequences for n jobs are defined by the schedule vector sequences for the three-job subsets of the n jobs, the schedule vectors of an n-job set will also cycle with a period equal to the processor schedule period of the n jobs.

For example, in the schedule given for the job set below the schedule vector repeats a t=20. This is the job set and the schedule vector sequence we formed in Section 3.9.

Job	Period	Execution Time
1	2	0.40
2	4	0.40
3	5	0.50



s_{12}	0	1	0	1	0	1	0	1	0	1	0
s_{13}	0	3	1	4	2	0	3	1	4	2	0
s_{23}	3	4	4	5	5	1	6	2	7	3	3

Figure 3-15 Three Job Valid Schedule

The definition of a schedule vector and the periodic nature of the tasks and the schedule element sequences for each job pair makes each of the vectors in the example descriptive of the same schedule. As seen in the example the schedule that is defined by

$$\begin{bmatrix} s_{12} \\ s_{13} \\ s_{23} \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix}$$

also contains the schedule vector

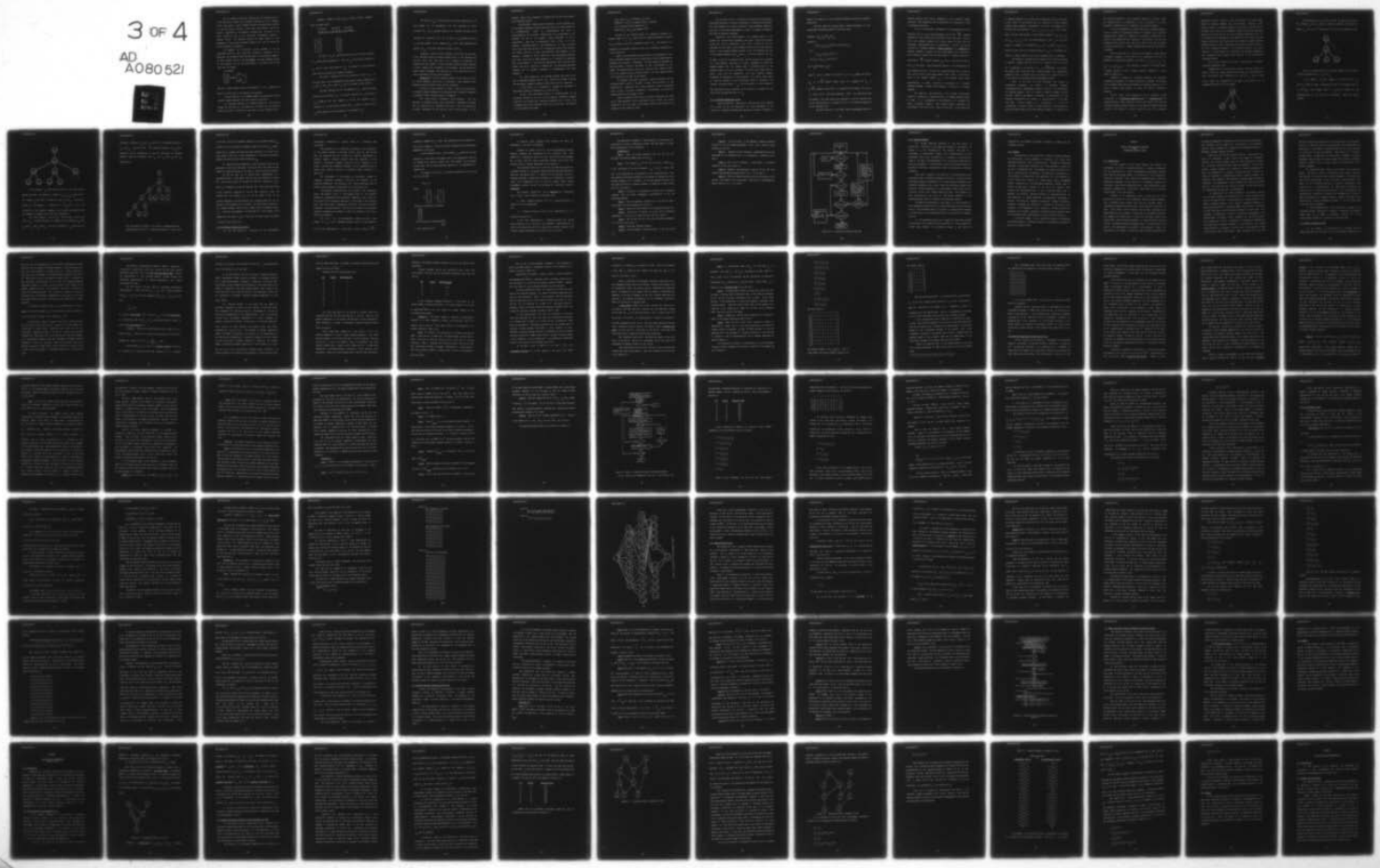
$$\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

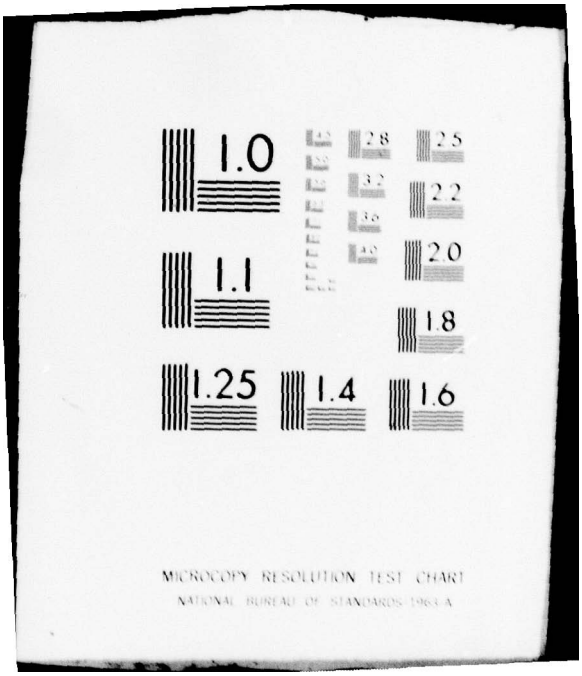
and vice-versa.

Each of the vectors in the sequence contains the schedule elements that define the relationship of each of the job pair's start times as though the schedule's reference time were defined to be at that point.

AD-A080 521 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
OPTIMAL MULTIPROCESSOR SCHEDULING OF PERIODIC TASKS IN A REAL-T--ETC(U)
DEC 79 W D SEWARD
UNCLASSIFIED AFIT/DS/EE/79-2 NL

3 OF 4
AD A080521




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

For any subset of jobs that contains the LCC reference job, or if any other job is chosen as the schedule reference job (i.e., assigned a start time equal to zero), schedule vectors are defined only relative to that job's active periods. On the other hand, when no job in the subset is specified as the schedule reference job, the start of an active period for each of the jobs under consideration defines a schedule vector. The schedule vectors are again defined by the sequences of the job pair's schedule elements, but only as long as the initial schedule is consistent.

For example, for a consistent initial schedule it can be determined which job that particular schedule is referenced to by examining the precedence relations specified by the schedule elements. The next schedule vector is the same <u>except</u> for those elements related to the reference job, as the relationship of the other jobs to each other will not change.

For example,

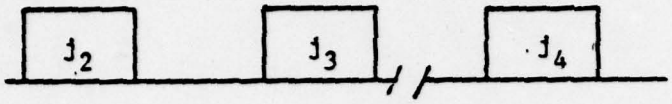
$$\begin{bmatrix} s_{23} \\ s_{24} \\ s_{34} \end{bmatrix} \text{ becomes } \begin{bmatrix} \text{new } s_{23} \\ s_{24} \\ \text{new } s_{34} \end{bmatrix}$$

when the initial schedule vector is referenced to job j_3 regardless of which is the next job in the reference job sequence.

The new values for the changed elements are determined from the schedule element sequences for their respective job pairs.

As originally defined, the schedule element for each job pair assumes the next value in its sequence at the "next" active interval of either of the jobs (unless one of the jobs is the LCC reference job).

Consider a subset of jobs j_2, j_3, j_4 with an initial schedule vector as shown below:



$$\begin{bmatrix} 0 \\ s_{23} \\ 0 \\ s_{24} \\ 0 \\ s_{34} \end{bmatrix} \qquad \begin{bmatrix} 1 \\ s_{23} \\ 1 \\ s_{24} \\ 1 \\ s_{34} \end{bmatrix}$$

The value of s_{24} as defined at the start of the active interval of j_4 shown above determines the value of s_{24} at the active interval of j_3, and it is the value shown as s_{24}^1. The reason for this should be clear from our difinition of schedule elements.

The initial reference job in the illustration above is j_2. So the next vector will be determined by the schedule element sequences for j_2 and j_3, and j_2 and j_4 while s_{34} will be the same in both vectors.

The next reference job in the sequence is j_3. Clearly in going from an active interval of j_2 to an active interval of j_3, the value of s_{23} changes to the next element in the job pair sequence $\langle s_{23} \rangle$. Likewise, if the next active interval were j_2 again, the next value of s_{23} would again be the next element in the sequence $\langle s_{23} \rangle$.

The value of s_{24}^{1}, no matter which is the next reference job (in this example j_3), is determined by the next occurance of active intervals of j_4 and j_2 whether either is the reference job next in the reference job sequence or not. So, the value of s_{24}^{1} assumes the value of the next element in the sequence $\langle s_{24} \rangle$. This same argument would applies to s_{23} if the next reference job were to be j_4.

Therefore, given an initial schedule vector that is consistent we can determine from the schedule elements which is the reference job for that vector. The next vector in the sequence is determined by assigning the next element in each individual schedule element sequence to those elements that correspond to the reference job and leaving all other elements unchanged. The reference job of the resulting vector can then be determined and the process repeated.

Theorem 3.10 Given an arbitrary subset of jobs from an LCC and an initial schedule vector that is consistent, every schedule vector generated from the initial vector will be consistent.

The proof of this theorem is a direct result of the definition of schedule element sequences, the techniques used to generate sequences of schedule vectors, and periodicity of schedules.

As a result, given a consistent initial schedule vector the sequence of schedule vectors can be determined, regardless of whether the LCC reference job is assigned a zero start time or no constraints are made concerning the job with the first active interval in the

schedule. Again, such a sequence of vectors will be cyclic and contain only consistent vectors.

The difference between sets that specify the LCC reference job as the schedule reference job and those that don't is a direct result of L1 inconsistencies. Since L1 inconsistencies are also L2 inconsistencies, for an initial schedule that is L2 inconsistent, it is possible for one of the schedule vectors generated to be L1 inconsistent. Once the L1 inconsistent vector occurs, it is not possible to determine which is the reference job. In fact, all three jobs of any three job set that is L1 inconsistent could be the reference job. Hence there are at least three possible next elements in the sequence. Although it can be shown that only inconsistent schedules will occur within any of these sequences, it is difficult to determine when every element of the particular equivalence class has been found unless each vector created is compared with all previously generated elements; and all possible sequences (or subsequences) have been examined.

For these reasons, we will consider forming partitions of the set of consistent schedule vectors only. We will not attempt to define any equivalence classes of schedule vectors which are not consistent.

At this point, it is possible to consider the existence of equivalence classes of schedules for a given job set.

We will form equivalence classes of schedules based on the following relation on the set of all possible consistent schedule vectors, X, for a given load consistent compatible set of n jobs. X is a nonempty set for any compatible set of two or more jobs. The relation * on X is defined as follows:

$S_A * S_B$, that is S_A is related to S_B by the relation *, if S_B is a schedule vector contained in the schedule vector sequence of S_A. Where both S_A and S_B are elements of X.

Because of the periodic nature of consistent schedules of periodic tasks, if $S_A * S_B$ then the set of schedules defined as related to S_A is equal to the set of schedules related to S_B. The relation * is therefore an equivalence relation on the set of consistent schedules for a given n-job LCC.

Based on the equivalence relation above we can determine for any given consistent schedule vector its equivalence class by generating its schedule vector sequence. Futhermore, the exact number of schedules in each equivalence class for any subset of jobs is determined by the number of reference points within the period of that subset of jobs. (i.e., The number of active intervals within the least common multiple of the periods of the jobs in the subset.)

For subsets which have a specified schedule reference job, the reference points occur at the start of each active interval of the reference job. There will be as many schedules in each equivalence class as there are active intervals of the reference job within the schedule period of the subset of jobs.

When the subset of jobs does not specify a reference job, there are as many schedules in the equivalence class as there are active intervals of all of the jobs in the subset within in each period of the schedule for that subset of jobs.

At this point we are in a position to exploit the relationships established previously to define an algorithm to determine the existence of a valid schedule for a given collection of load consistent compatible jobs, and to use these relationships to reduce the number of schedules that must be explicitly examined.

Because for any valid schedule, the reference point of the schedule can be arbitrarily assigned to the start of any active period of any job, and, in particular, to an active interval of the LCC reference job; and since in each schedule there must occur each possible value of every schedule element relative to the LCC reference job, one of the schedule elements, say s_{ik} where j_i is the LCC reference job, may be fixed at any of its possible values and not preclude the occurrence of a valid schedule. Naturally, it is most advantageous to choose the schedule element with the largest range of possible values as the schedule element to use as a reference. The exact value chosen does not matter since only its combination with other schedule elements determines the existence of a valid schedule. As a result, there exists at least one element in each equivalence class of schedules that contains the given value of s_{lk}, and the validity of every element in each equivalence class is implied by the validity of a schedule for the vector with the specified value of s_{lk}.

3.12 Consistent Enumeration Trees

Before we define an algorithm for constructing a valid schedule for a given LCC, we will consolidate our prior development of the concepts of consistent schedules and equivalence classes of schedules to

enable us to reduce the set of possible schedules that must be examined explicitly.

We state again the general mixed integer formulation of the uniprocessor scheduling problem for periodic jobs:

$$\text{maximize} \quad f(\bar{s}_{ik}, \bar{t}_{0i}) = \bar{c}_1 \bar{s}_{ik} + \bar{c}_2 \bar{t}_{0i}$$

subject to

$$E_i - T_i \leq t_i - t_{0k} - s_{ik} \, CI(i,k) \leq CI(i,k) - E_k - T_i$$

and

$$0 \leq s_{ik} \leq (T_k + T_i)/CI(i,k) - 1$$

for all j_i and j_k in the LCC, K,

and $\bar{s}_{ik} \geq \bar{0}$ integer, $\bar{t}_{0i} \geq \bar{0}$

where $\bar{c}_1$ is an n_1 element row vector; $\bar{c}_2$ is an n_2 element row vector; $\bar{s}_{ik}$ is an $\binom{n}{2}$ element column vector of integers; and $\bar{t}_{0i}$ is an $\binom{n}{2}$ element column vector of nonnegative real numbers. The vectors $\bar{c}_1$ and $\bar{c}_2$ are at this point undefined. (Note, this formulation does not assume a job start time of zero assigned to the LCC reference job, or the specification of a constant for one of the schedule elements of the LCC reference job.)

The exact solution of the mixed integer programming problem, in

general, requires some form of enumeration of the possible integer solutions. This enumeration can be represented as a traversal of a tree as discussed previously.

In all of the previous development, it was assumed that for the particular LCC of n jobs being considered, each of the $\binom{n}{2}$ schedule elements for that collection was available at the same time. That is, all of the schedule elements of a given subset of jobs could be considered simultaneously and not sequentially. We will call such a collection of schedule elements a <u>closed collection</u> or a <u>closed set</u> of schedule elements. Likewise, for any m-job subset of an n-job LCC, a set of schedule elements for the m-job subset is closed if and only if there exists $\binom{m}{2}$ schedule elements s_{ik}, one for each pair of jobs j_i and j_k in the subset. This terminology relates to the representation of the schedule vector of an LCC as an undirected graph with a node for each job in the set and an edge between nodes i and k if and only if there is the schedule element s_{ik} in the schedule vector under consideration. A closed collection is equivalent to a strongly connected subgraph of nodes which represent a subvector of schedule elements.

The reason for our restriction of the concept of equivalence classes of schedules to closed collections of schedule elements should be clear. It is a consequence of the definition of a schedule as a vector of schedule elements. Each schedule element represents the relationship of two job's start times to each other. If the collection

of schedule elements is not closed, the relationship of two or more jobs to each other is not defined and the subvector of those schedule elements cannot represent a subschedule of the LCC. For example, for a four-job LCC, the collection $\{s_{12}, s_{13}, s_{23}, s_{14}, s_{24}\}$ is not closed. There are two closed subsets of the schedule elements $\{s_{12}, s_{13}, s_{23}\}$ and $\{s_{12}, s_{14}, s_{24}\}$, but the relationship of j_3 and j_4 to each other cannot, in general, be infered from s_{13} and s_{14} and/or s_{23} and s_{24} except to bound the set of possible values of s_{34} to a consistent set. Without a closed collection, part of the schedule vector is undefined.

The problem of insuring that we consider only closed collections of schedule elements does not occur if the schedule vector for a given LCC is examined in its entirety, i.e., all $\binom{n}{2}$ elements at one time. The problem does arise when the schedule elements are sequentially examined to form a schedule vector one element at a time as we will do in our solution of the mixed integer programming problem.

Also, at the time we first introduced the concept of inconsistent schedules, our ability to determine schedule inconsistencies was dependendent on what we now know to be closed sets of schedule elements. That is, triples of schedule elements in which each pair implies the schedule consistency or inconsistency of the schedule represented by the triple. It was pointed out at the time, that when a given schedule is L1 inconsistent it is also L2 inconsistent; and there exists a simple inequality which relates the upper and lower bounds for the third schedule element of a closed set to

the other two elements if an L2 consistent schedule is to exist. Since any schedule that is L2 consistent is also L1 consistent, it is only necessary to determine the L2 consistency of any given schedule.

It is this dependence on closed collections of schedule elements for our concepts of equivalence classes of schedules and consistent schedules which makes the ordering of the schedule elements and the levels of the enumeration tree important. Our objective is to form the enumeration tree in such a way that at any level of the tree the number of closed collections of schedule elements which have been specified is a maximum.

While no proof will be given here, the number of closed subsets of schedule elements at any level of the enumeration tree will be maximized by ordering the elements as follows: $s_{li}, s_{lk}, s_{ik}, s_{lp}, s_{ip}$, $s_{kp}, s_{lr}, s_{ir}, s_{kr}, s_{pr}, \ldots$ This ordering is related to the minimal number of edges required to form a strongly connected subgraph of a given number of nodes $m \leq n$.

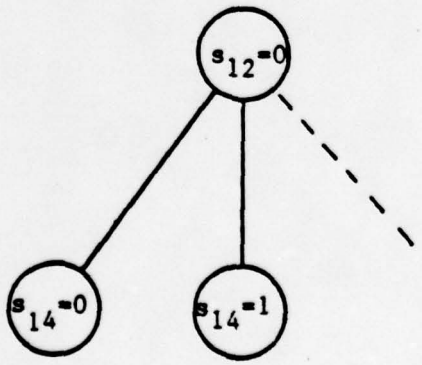
Because our concepts of consistent schedules and equivalent schedules are dependent on closed collections of elements, the ordering of the elements shown above will maximize the instances where we will be able to employ these concepts to reduce the explicit enumeration required.

We will refer to the enumeration tree of consistent closed collections as a <u>consistent enumeration tree</u> or <u>consistent tree</u> for shorter notation. Such a tree will have at each level of the tree only consistent schedule elements relative to the schedule elements specified at each of the levels of the tree nearer the root. The consistent

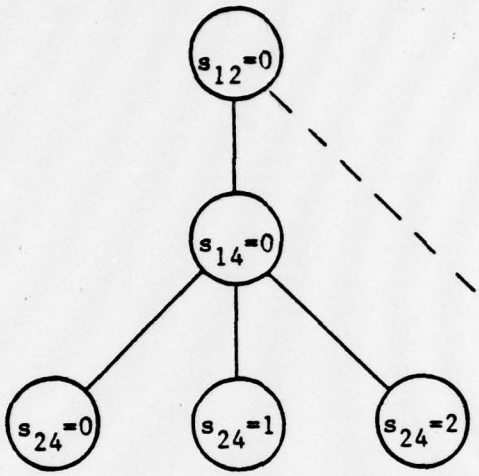
enumeration tree for any set of jobs will contain only those closed subsets of schedule elements that are consistent. Moreover, every closed collection of schedule elements for the entire LCC that will be contained in the tree will be consistent schedules; and every consistent schedule for the job set will be in the consistent enumeration tree. We will ensure this by constructing the enumeration tree a level at a time adding at each level only those nodes that represent consistent schedule elements relative to those nodes previously defined at higher levels of the tree. Therefore, because it is a necessary condition for a given schedule to be valid that it be consistent, the consistent enumeration tree represents every possible valid solution to the uniprocessor scheduling problem.

We will now illustrate the step by step process of forming a consistent enumeration tree.

Suppose there is a four-job set and the first two nodes of the branch of the tree being considered are ordered as illustrated below. There are possibly other branches of the tree for s_{14} equal to other values. These are represented by the dashed line and will not be represented explicitly in this example.

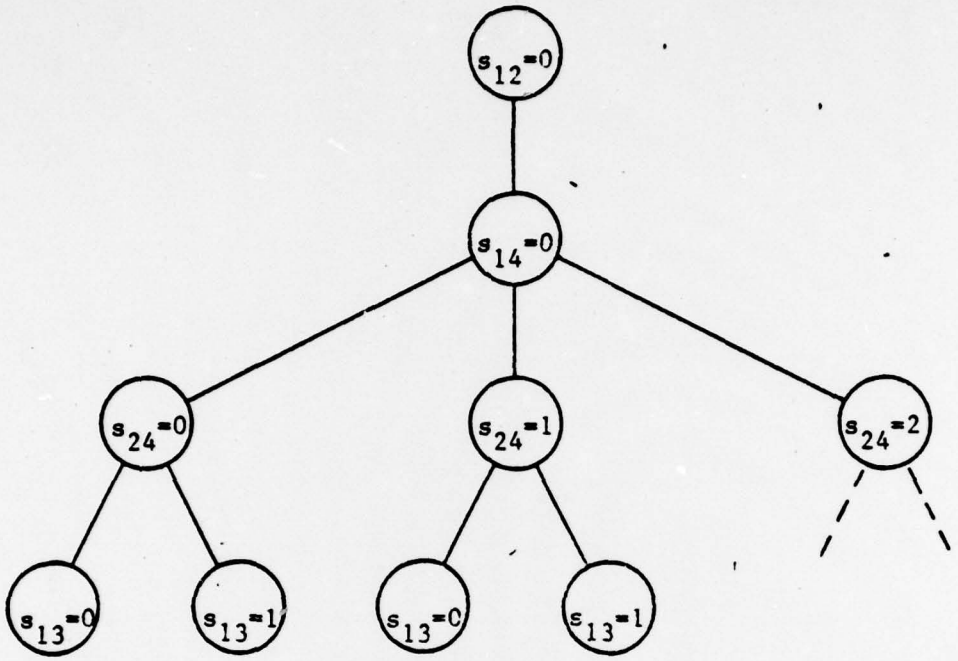


The characteristics of the job set and the specified values of the elements s_{12} and s_{14} define a range of consistent values for the element s_{24}, say {0,1,2}. This branch of the tree now assumes the form



This is a closed collection of schedule elements and can specify consistent schedules for j_1, j_2, and j_4.

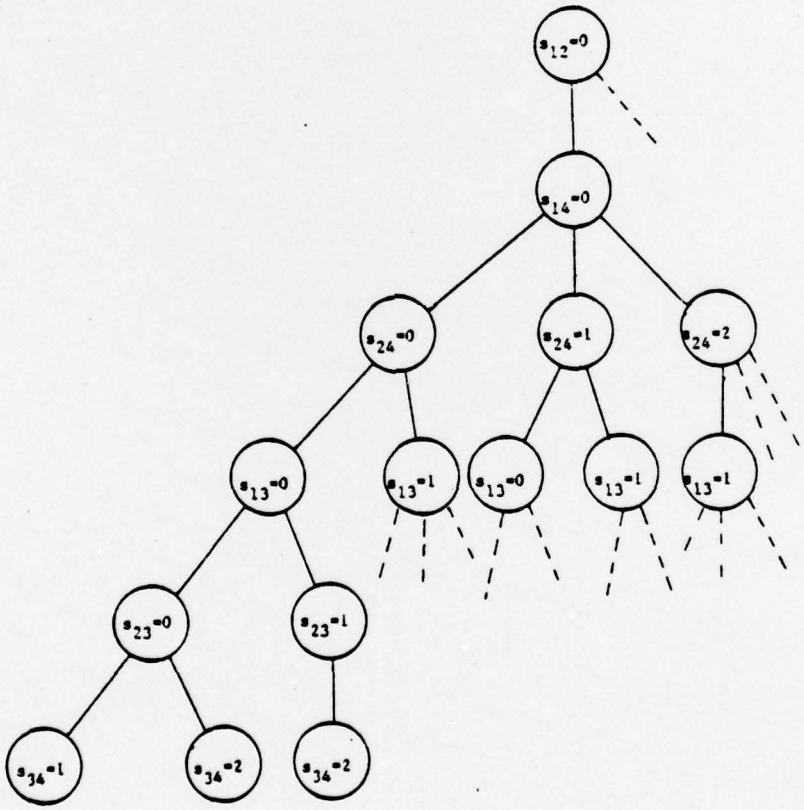
In our example, the next element in the tree will be s_{13}. Since s_{13} forms no closed collections with any of the elements above it in the tree, every integer value of s_{13}={0,1} as defined by the characteristics of the job set are consistent. Hence the branch becomes,



For the element s_{23} which we will now add to the tree, we will expand only about the sequence of elements $\langle s_{12}, s_{14}, s_{24}, s_{13} \rangle = \langle 0,0,0,0 \rangle$. The element s_{23} will form a closed set with s_{12} and s_{13}, and we will assume, for this example, a consistent set of $s_{23} = \{0,1\}$. But, the collection of five schedule elements is not closed and therefore does not represent a schedule of the four jobs represented.

The final element of this job set and the final level of the tree is s_{34}. Closed collections of elements will be formed by s_{34} with s_{13} and s_{14}, and s_{23} and s_{24}. The set of elements of s_{34}, which will be

consistent relative to s_{13} and s_{14}, may not be consistent relative to s_{23} and s_{24}, and vice versa. The consistent elements for s_{34} will, therefore, be the intersection of the two collections of schedule elements which are consistent with s_{13} and s_{14}, and s_{23} and s_{24} respectively.



The final graph of a subset of the nodes is represented above. Generalizing the notion of consistent elements at a given level

of the tree, the set of consistent elements for any schedule element s_{km} relative to all of the pairs of schedule elements with which s_{km} forms a closed set of elements and which have been specified at a level of the tree nearer to the root is the intersection of the sets of consistent elements from every closed triple containing s_{km}.

By constructing the enumeration tree as defined above, we are guarenteed to have only consistent collections and at the same time the number of closed collections at any level of the tree will be maximized.

We will now make use of another result demonstrated in a preceding section. It was shown that for any valid schedule of a given LCC, there occurs within the schedule at least one instance of $s_{li}=0$, where j_1 is assumed to be the LCC reference job. Each equivalence class of the consistent schedules for the LCC must contain at least one schedule with the element $s_{li}=0$. So, we can assign to any of the schedule elements of the LCC reference job a constant equal zero and not prevent a valid schedule from occurring. We could assign a constant to the schedule element which has the largest range of possible values, s_{lk}. With such an assignment, we would make the first element in the enumeration tree equal to $s_{lk}=0$. This would in effect reduce the number of levels in the tree by one.

3.13 An Optimal Scheduling Algorithm

We will now define our algorithm for the nonpreemptive

uniprocessor scheduling of periodic tasks in a hard-real time environment.

The optimality of our algorithm, as defined in the introduction to this chapter, depends on two main points. First, we must be sure that our algorithm will not exclude any possible combinations of schedule elements (schedule vectors) which may represent a valid schedule. The second necessary condition for our algorithm to be optimal involves the systematic traversal of the enumeration tree to insure every possible solution is considered either explicitly or implicitly.

Our development of the concepts of equivalence classes of schedules, consistent schedules, and finally the formation of the consistent enumeration tree assures us that every equivalence class of possibly valid schedules is represented by the consistent tree.

The algorithm presented below will consider each subschedule vector and schedule vector in lexicographic order of the schedule elements as represented by the consistent enumeration tree. While not all vectors will be considered explicitly, at any stage of the algorithm every vector with lower lexicographic order will have been considered previously. The algorithm will make use of the concepts of equivalence classes of schedules and fathoming to make the traversal of the tree as efficient as possible.

Let I be a list of lexicographicly ordered vectors. Each element of I, I_m, is an infeasible schedule vector of the consistent tree for the subschedule of r jobs, where r<n and there are $\binom{r}{2}$ -1

nonnegative elements in I_m. Each node represented by each element of I_m will also be assumed to have not yet been fathomed by the lexicographic enumeration of the tree.

In each schedule vector, every element, s_{ik}, which has not been specified at this level or any higher level of the enumeration tree will be defined to be equal to a minus one. Each element in the schedule vectors which has not been previously defined will be represented by a negative one.

For example, let the set of infeasible schedules for a four job LCC be as follows:

$$I=\{I_1, I_2\}$$

where

$$I_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 2 \\ 1 \\ 1 \end{bmatrix} \qquad I_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$$

defines as infeasible the schedule

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 2 \\ 1 \\ 1 \end{bmatrix}$$

and the three-job subschedule $\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

of the consistent tree.

In addition, every schedule which contains the three job subschedule I_2 will also be infeasible.

Likewise, we define a list F to be the lexicographicly ordered feasible schedules and subschedules of the consistent tree. Each element of F represents a node of the tree which has not been fathomed but is valid. Once the node is completely fathomed and is valid, the schedule vector it defines represents a valid schedule.

The procedure we will use to solve the job scheduling problem is a modified version of the general branch-and-bound algorithm defined by Geoffrion and Marsten (Geo). In Section 3.8, we discussed the use of candidate problems, relaxation of problems, and fathoming of a problem (node) of an enumeration tree to solve the general mixed integer programming problem. We will now introduce the additional concept of separation.

Any candidate problem (CP) can be separated into subproblems $(CP_1),..(CP_r)$ if the following two conditions hold:

1. Every feasible solution of CP is a feasible solution of exactly one of the subproblems.

and

2. A feasible solution of any of the subproblems CP_i is a feasible solution of CP.

In our case, separation of a candidate problem will entail partitioning the CP at each node into decendent subproblems each of which is the restriction of CP to a particular consistent integer of the schedule element represented by the next level of the tree.

The algorithm to determine a valid schedule, if one exists, for a given LCC of n jobs is defined as follows: (The next figure is a flow chart representation of the procedure).

Algorithm 3.1

Step 1. Form the linear constraint sets for the LCC and determine the schedule element sets of each s_{ik}.

Step 2. Find $s_* = \max\{:s_{ik}: \text{ for all } i<k \text{ in the LCC }\}$, where $:s_{ik}:$ is the cardinality of the set of elements s_{ik}. Set $s_* = 0$. (Note: This step of the algorithm is not essential to the problem solution. This step enables the possible schedules to be reduced by defining one of the schedule elements to be a constant. For the more general solution, skip this step and allow all schedule elements to assume any of their values which are consistent.)

Step 3. Initialize a lexicographicly ordered list of candidate problems with the linear programming representation of the job scheduling problem.

Step 4. Test the candidate problem list. If the list is empty, stop; there is no valid schedule for this LCC.

Step 5. Select the first problem in the candidate problem list.

Step 6. Test to see if this CP is in the list of infeasible schedules I. If CP is in I, remove CP from I and go to Step 4.

Step 7. Test to see if CP is in the list of feasible schedules. If CP is in F, go to Step 11.

Step 8. Solve the candidate problem.

Step 9. Is the candidate problem feasible? If not, go to Step 12.

Step 10. Is the solution to the candidate problem a feasible solution to the job scheduling problem. If so, stop. This is a valid schedule for the LCC.

Step 11. Separate CP into consistent subproblems and add each subproblem to the candidate list. If no separation is possible, go to Step 4.

Step 12. Does the CP represent a closed subset of schedule elements? If not, go to Step 14.

Step 13. Generate the equivalence class of CP and add each schedule lexicographicly greater than CP to I. Go to Step 4.

Step 14. Find the largest closed set which is an ancestor of CP and add each element of its equivalence class which is lexicographicly greater than CP to F. Go to Step 4.

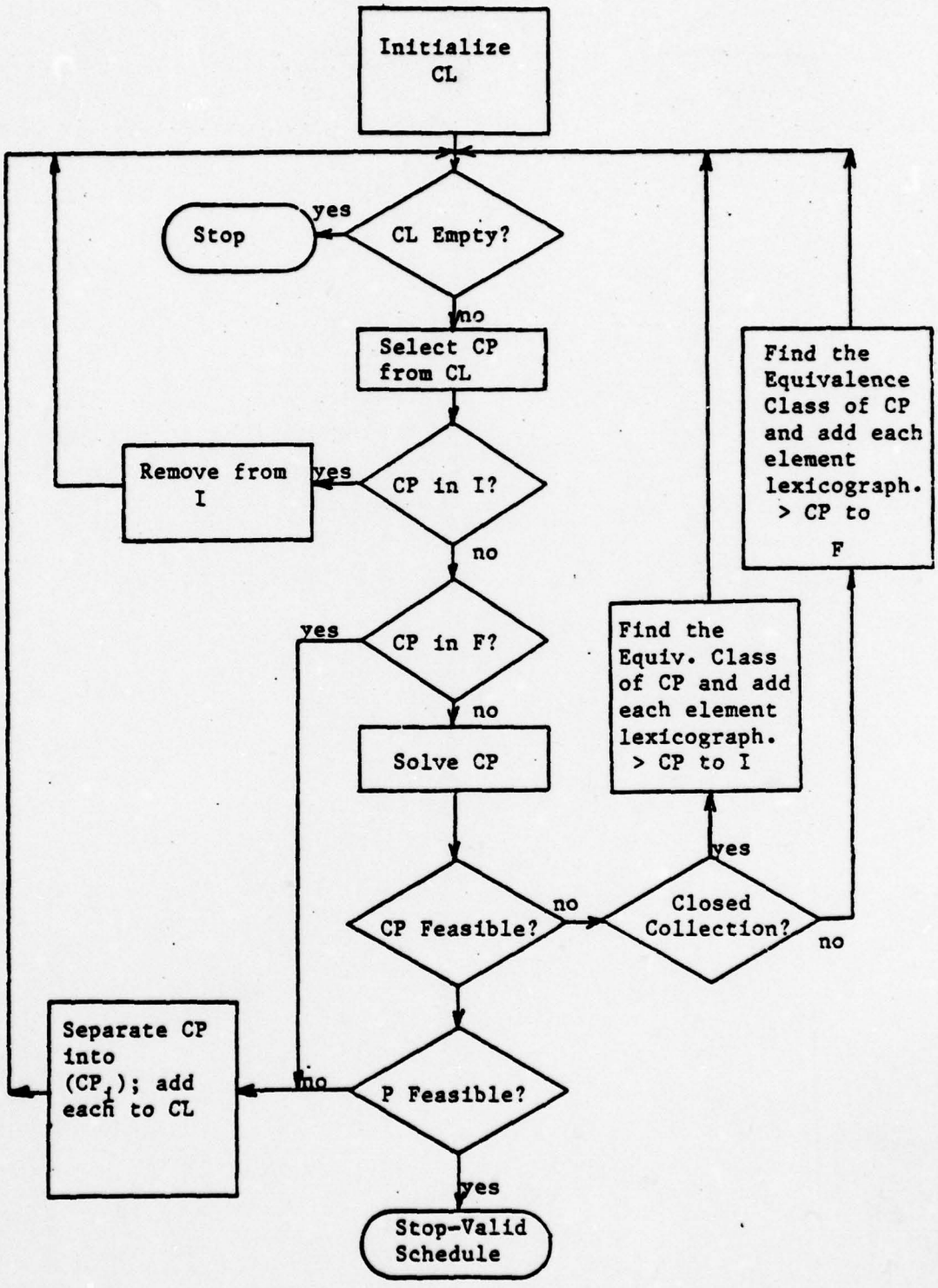


Figure 3-16. An Optimal Scheduling Algorithm

3.14 Optimal Schedules

The optimal algorithm presented in the last section is specifically structured to find any valid schedule if one exists. As alluded to in the introduction to this chapter, there may be a requirement to differentiate between valid schedules based on some performance metric such as sensitivity to variation in job exicution times or the ease of reconfiguration of the pariticular system given a given job assignment. In this section, we will very briefly discuss the extension of the above algorithm to include the construction of optimal schedules.

Every valid schedule of the given LCC is contained within some equivalence class of feasible schedules for the LCC, and each of those equivalence classes is represented by a valid schedule contained in the consistent enumeration tree above. While the equivalence classes above have been defined independent of any performance metric other than the inclusion of each valid schedule within any other valid schedule, the above algorithm will also serve to find optimal schedules if the performance metric does not redefine the equivalence classes.

To modify the algorithm for such cases, it is only necessary to add a fathoming criteria which will fathom nodes based on the achievable value of the performance metric if the node is separated when compared to a known value of the performance metric for some partial or complete solution.

If the performance metric used changes the equivalence classes to partitions of the classes of valid schedules, it is only necessary to include those elements of the schedule element s* that were not

considered in the original algorithm in addition to adding the new fathoming criteria.

3.15 Summary

The objective of this chapter was to define an optimal algorithm for constructing a valid schedule, if one exists, for a given Load Consistent Compatible (LCC) of periodic jobs. We began by determining necessary and sufficient conditions for a valid schedule to exist for any pair of compatible jobs. We extended this result to define for an LCC of arbitrary size the necessary and sufficient conditions for a valid schedule. We showed that the construction of a valid schedule for an LCC involved the solution of a mixed integer linear programming problem formulation of the uniprocessor scheduling problem. Next, we defined an equivalence relation on the set of all possible valid schedules and demonstrated that it was a relatively simple task to generate for any given consistent schedule its equivalence class. We then formulated an optimal algorithm for uniprocessor scheduling of periodic jobs using the concepts of equivalence classes of schedules, consistent schedules and relaxation to improve the efficiency of the traversal of the enumeration tree. Finally, we briefly discussed the extension of the optimal algorithm to determine optimal schedules based on a performance metric expressible as a linear objective function.

In the next chapter, we will address the problem of determining the minimal number of processors required to nonpreemptively schedule a given set of periodic jobs.

CHAPTER 4

MINIMAL MULTIPROCESSOR SCHEDULES FOR INDEPENDENT PERIODIC TASKS

4.1 Introduction

In this chapter, we will bring together the concepts and developments of the two preceeding chapters to formulate an algorithm for the construction of a static nonpreemptive multiprocessor schedule which requires the minimal number of processors for a valid schedule of a given set of periodic jobs. While only independent sets of tasks are specifically addressed in this chapter, the scheduling of general sets of periodic tasks can be accomplished with a small modification to the algorithm presented here. The required modification will be discussed in more detail in the next chapter.

Although it is conceptually easy to form and test every partition of a given job set to determine a minimal processor schedule, such a procedure could require more than the available amount of computational resources and may in fact not be feasible for some sets of jobs. Such a "brute force" technique does not take advantage of the characteristics of the problem to reduce the set of possible solutions. For example, the classification of jobs into collections which do not exclude each other from being assigned to a single processor, when the jobs are considered pairwise, eliminates some of the partitions of the job set from consideration as possible valid schedules. Hence, our discussion in this chapter is directed toward defining an algorithm for

the implicit enumeration of partitions of the job set that may specify a valid schedule. We will make use of the structure of the problem to reduce as much as possible the computational requirements necessary to determine a minimal processor schedule.

Toward this end, we will initially develop a theory which will allow us to determine a lower bound on the number of processors required. This bound is based on both the load constraints of a given set of jobs and the sets of jobs which are compatible. We will then develop a formal technique for ordering the possible combinations of valid job schedules into a lattice structure which will enable us to define an algorithm to systematically examine possible schedules and construct a valid schedule which requires a minimum number of identical processors. The combination of jobs considered will include only those which are LCC's and cover the job set, as these are the only possible job combinations that may have a valid schedule.

As specified in the formal statement of the problem, (Section 2.2), any valid schedule is a paritition of the job set such that each block of the partition has a valid schedule on a single processor. The minimal processor schedule can be found by successive examination of every partition with ever increasing number of blocks until a partition is found which has a valid uniprocessor schedule for each block. This technique is inefficient in view of our previous developments on the classification of periodic tasks. Therefore, the initial sections of this chapter will focus on techniques of systematically forming only those partitions which may represent a valid schedule, also forming these partitions in a sequence such that once a valid schedule is found all partitions which would require fewer processors will have been

previously examined and eliminated as not representing a valid schedule. Finally, an additional feature of the algorithm that defines upper and lower bounds on the number of processors required for a valid schedule of a given job set will be discussed. These bounds will be shown to converge toward a common value as the procedure converges on a optimal assignment, thus allowing the system designer the freedom of choosing a suboptimal schedule which is within known worst-case bounds of the optimal assignment.

The algorithm we will define in this chapter has two distinct phases. The initial phase of the algorithm will provide a means of partitioning the job set so that certain assignments can be made which will allow a partitioning of the job set into smaller subsets. This phase will not provide any reduction in required computation in some cases and can be eliminated from all cases without preventing the construction of a minimal processor schedule. The second phase of the algorithm will structure the possible job assignments and provide a means of searching for a minimal processor schedule. This phase of the algorithm will function regardless of whether or not the initial phase of the algorithm is employed.

In the following section, we will extend the concepts of load consistent maximal compatibles to formulate a procedure to determine a lower bound on the number of processors required for a valid multiprocessor schedule for a given set of periodic jobs.

4.2 Minimal Coverings of LCMC

In the chapter on classification of periodic jobs, we demonstrated that the collection of all of the maximal compatibles of a

given job set, K, represents all of the possible combinations of those jobs which do not exclude one another pairwise. It was also pointed out that for a valid schedule to possibly exist for a given collection of jobs, which are all elements of the same maximal compatible, the sum of the load factors of those jobs must not exceed unity. In addition it was noted that the collection of all of the Load Consistent Maximal Compatibles (LCMC's) of a given job set consists of every subset of the maximal compatible that is load consistent and is not contained in a larger load consistent subset. The collection of LCMC's of a given job set specify all of the possible combinations of jobs that do not exclude each other pairwise or exceed the execution load constraint of a single processor.

A collection of subsets $C=\{A_i\}$ of an arbitrary set A is said to cover A if for every element a_k of A there exists at least one subset A_j in the collection of subsets C that contains a_k (Gar).

We know that in order to schedule the job set, we must assign a job start time to each job in the set - hence, for a valid schedule, any collections of subset being considered must cover the job set. Furthermore, because each LCMC of the job set contains every possible combination of those jobs which do not exclude each other from forming a valid uniprocessor schedule, each LCMC represents a "maximal" possible assignment of jobs to a processor. Hence, we intuitively look to the smallest collection of LCMC's which contain every job of the job set for a lower bound on the number of processors required for a given set of jobs.

The problem of determining a minimal number of subsets of a collection of subsets that will cover a given set has been studied extensively under the title of the Set Covering Problem (SCP). Some of the applications of the covering problem include airline crew scheduling, simplification of boolean expressions, and vehicle scheduling (Chr, Gar).

The SCP derives its name from the following set-theoretic interpretation. Given a set $R=\{r_1, r_2, \ldots, r_n\}$, and a family of subsets $F=\{S_1, S_2, \ldots, S_m\}$ of R, then any subfamily $F_i=\{S_i, S_{i1}, S_{i2}, \ldots, S_{ik}\}$ of F such that

$$\bigcup_{j=1}^{k} S_{ij} = R$$

is called a set-covering of R, and the S_{ij} are called covering sets. If, in addition, each of the S_{ij} in F_i are pairwise disjoint, then F_i is called a set partitioning of R.

In general, there can be associated with each subset S_i of F a positive cost c_i. The SCP is to find that set-covering of R which has a minimum cost, where the cost of F_i is $\sum_{j=1}^{k} c_{ij}$. (Chr)

A set-covering, F_i, of a set R is a minimal covering if there is no covering F_j of R which contains fewer subsets S_i of F. A minimal

covering is a minimal cost covering when the cost c_i associated with each of the subsets S_i of F are equal.

In the next theorem, we will show that a minimal covering of LCMC's establishes a lower bound on the number of processors required for a valid multiprocessor schedule of a given set of periodic jobs. First though, we will examine the relationship of the lower bound on the number of processors to the least integer greater than or equal to the total load factor for a job set, which we will represent by LF(K), and the cardinality of a minimal covering of maximal compatibles of a job set K, $MC^{\#}(K)$.

In a previous chapter, it was shown that the number of processors required for a given job set could not be less than LF(K). This represents the execution time load requirements for the job set without any consideration of job compatibilities. Also, it was pointed out that since the maximal compatibles of a job set represent the largest collections of subsets of jobs which are pairwise compatible, there could be no valid schedule that requires fewer than $MC^{\#}(K)$ processors for the job set K. The coverings of maximal compatibles bounds the number of processors as a function of the compatibility of the jobs with no consideration of execution loading of the jobs (except for each pair within a maximal compatible). Therefore, the minimal number of identical processors required for a valid schedule for a given job set K can not be less than the maximum of LF(K) and $MC^{\#}(K)$. We further constrain this bound by citing examples of valid schedules which

occur for some cases where the number of processors required equals the maximum of LF(K) and $MC^{\#}(K)$.

Consider first the job set shown below

Job	Period	Execution Time
j_1	2	1
j_2	2	1
j_3	3	1
j_4	3	1
j_5	5	1
j_6	5	1
j_7	7	1
j_8	7	1

The total load factor of the job set is 2.35238. Each job is compatible with one and only one other job in the set. There are four maximal compatibles in the minimal covering each of which represents a valid schedule of two jobs. The minimal processor schedule requires $MC^{\#}(K)$ processors.

On the other hand, suppose all of the jobs in a set K are pairwise compatible (K has a single maximal compatible), and there exists no subset of K of three jobs that is load consistent. Then any partition of K into [n/2] subsets, where [A] represents the least integer greater than or equal A, each subset with two or fewer jobs, represents a valid multiprocessor schedule of K that requires [n/2] processors. There is no valid schedule which requires fewer processors.

Therefore, the minimal processor schedule of this job set requires LF(K) processors.

Finally, consider the job set below which has a total load factor equal to two and the are two maximal compatibles which cover the job set.

Job	Period	Execution Time
j_1	1	0.75
j_2	1	0.25
j_3	2	1.2
j_4	2	0.8

As the preceding examples illustrate, a lower bound on the minimal number of processors required for a valid schedule of a job set is $\text{Max}\{LF(K), \overset{\#}{MC}(K)\}$. But this bound can be made tighter as the following theorem shows.

Theorem 4.1 The minimal number of identical multiprocessors required for a valid schedule of a given set of periodic jobs, K, is greater than or equal to $\overset{\#}{LC}(K)$, where $\overset{\#}{LC}(K)$ is the cardinality of a minimal covering of LCMC's of K.

Proof As a result of the definitions of a minimal covering and LCMC's, there can be no fewer than the least integer greater than or equal to the sum of the load factors of all of the jobs in K in any covering of LCMC's. Likewise, any covering of LCMC's must not be less than any minimal covering of maximal compatibles of a given job set. Thus, any covering of LCMC's is greater than or equal to the maximum of $LF(K)$ and $\overset{\#}{MC}(K)$.

Each of the previous examples represents a valid schedule in which the minimal number of processors is equal to the cardinality of a minimal covering of LCMC's of K.

Suppose that the minimal covering of LCMC's is strictly greater than $MAX=Max\{LF(K), MC(K)\}$, and there exists a covering M which is not less than MAX but is less than the minimal covering of LCMC's. Suppose also that there is a valid schedule possible for the covering M.

Any valid schedule of K must be a partition of K such that each block of the partition consists of a subset of pairwise compatible jobs that are load consistent. Hence, a valid schedule from the covering M must be a partition of K in which each block must be an LCC of K. But, each LCC of K is contained in an LCMC of K. Thus there exists a covering of LCMC's of K which contain the partition obtained from M, and M cannot be less than a minimal covering of LCMC's. QED

In most applications of the SCP, it is sufficient to find one cover that minimizes the cost. But, while we are definitely interested in a minimal cover of LCMC's for a given job set to establish a lower bound on the number of processors required, we are not assured that for any given minimal covering there exists a valid schedule. We must, therefore, find every minimal covering. Furthermore, we are not assured that any collection of LCMC's which contain a minimal covering contains a valid schedule. It can be shown that it is possible for irredundant covers to exist which are not minimal covers (Pra).

We say that a cover of LCMC's of a job set K, $C_L(K)$, is an irredundant covering of K if the removal of any one of the LCMC's

contained in C_L causes C_L to no longer be a cover. That is, the removal of any LCMC, L_l, results in the removal of at least one job, j_i, of K from the collection C_L (Pra).

Hence, in addition to the minimal coverings, we must find all of the irredundant covers that are not minimal. (Clearly each minimal cover is irredundant.) There are certain a-priori deductions and reductions which are possible for the general SCP, but instead of addressing the reduction of the SCP in general, we will now consider those which are specific to the problem of finding all of the irredundant coverings of LCMC's of a given set of periodic jobs.

A <u>cover table</u> of LCMC's for a given job set K of n jobs is an r by n matrix such that there are r rows one for each LCMC and n columns one for each job j_i in K; and there exists a one in column m and row p if and only if the job j_i of K represented by column m is contained in the LCMC represented by the row p. The cover table represents each and every covering of a given job set. We further define a <u>reduced cover table</u> of a job set K as a sub table of the job set cover table from which some rows or columns or both have been eliminated.

The reduction rules we will now define are based on the cover table of the job set and the the requirement that we find <u>all</u> of the irredundant coverings of LCMC's for the job set.

Since there exist for each job in the set at least one LCMC that contains that job, there exists at least one covering of K from the set of all LCMC's of K.

Rule 1. If there exists some job j_k of K such that j_k is contained in an LCMC, L_i, and j_k is contained in no other LCMC of K, then L_i must be in all solutions and the problem may be reduced by eliminating all j_m contained in L_i from the table. Such an LCMC, L_i, is defined to be an essential LCMC of the job set K.

Rule 2. If there exists within a reduced cover table rows that are equal, form the sets of equivalent rows of the table and eliminate all but one row from each equivalence class of rows. In the final solution of the problem each of these rows represent solutions to the covering problem. There can never be equal rows in the original cover table due to the definition of LCMC's so if there are no essential LCMC's then this rule will not apply.

Rule 3. Examine the reduced table formed above for "essential" LCMC's of the reduced problem as done in rule 1.

Rule 4. When the cover table can be reduced no further by application of rules 1 thru 3, the final solution must be formed as a boolean function of products of sums of LCMC's required to cover the remaining jobs. The technique of solution by means of a tree structure is similar to that of determining the set of Maximal compatibles as shown in Chapter 2.

To illustrate the process of determining all of the irredundant covering of LCMC's consider the collection of LCMC's for the example job set of Chapter 2.

$L_1 = (j_1, j_1, j_5, j_6, j_{10})$

$L_2 = (j_2, j_5, j_6, j_7, j_{10})$

$L_3 = (j_3, j_6, j_7, j_9)$

$L_4 = (j_4, j_2, j_6, j_9)$

$L_5 = (j_5, j_6, j_8, j_9)$

$L_6 = (j_6, j_4, j_6, j_9)$

$L_7 = (j_7, j_3)$

$L_8 = (j_8, j_2, j_5, j_6, j_{10})$

$L_9 = (j_9, j_1, j_2, j_{10})$

The cover table is

	1	2	3	4	5	6	7	8	9	10
L_1	1	0	0	0	1	1	0	0	0	1
L_2	0	0	0	0	1	1	1	0	0	1
L_3	0	0	0	0	0	1	1	0	1	0
L_4	0	1	0	0	0	1	0	0	1	0
L_5	0	0	0	0	0	1	0	1	1	0
L_6	0	0	0	1	0	1	0	0	1	0
L_7	0	0	1	0	0	0	0	0	0	0
L_8	0	1	0	0	1	1	0	0	0	1
L_9	1	1	0	0	0	0	0	0	0	1

The essential LCMC's are L_5, L_6, and L_7. Each of these LCMC's will occur in every covering of K.

The reduced table is

	1	2	5	7	10
L_1	1	0	1	0	1
L_2	0	0	1	1	1
L_3	0	0	0	1	0
L_4	0	1	0	0	0
L_8	0	1	1	0	1
L_9	1	1	0	0	1

Note for the reduced table, L_2 covers more than L_3 and contains L_3, but for the original table inclusion of L_2 in any cover does not imply the inclusion of the elements of L_3. Therefore L_3 cannot be eliminated from the reduced table. This is a consequence of searching for all of the irredundant covers and not just a minimal covering.

In the reduced table above, there are no rows which are equal. If there were we would eliminate all but one of those rows since they would be equivalent in any covering of the reduced table. In the final solution of the irredundant coverings of the original table, every irredundant covering of the reduced table would be combined with the essential LCMC's of the original table and all combinations of the equivalent coverings of the reduced table would be created.

There is no essential LCMC for the reduced table so we will now form a boolean expression of LCMC's which cover each job of the reduced table

$$f(L)=(L_1+L_9)(L_4+L_8+L_9)(L_1+L_2+L_8)(L_2+L_3)(L_1+L_2+L_8+L_9)$$

The irredundant covers which result when the essential LCMC's are combined with the expansion of the above boolean function are:

{L ,L ,L ,L ,L }
  5 6 7 2 9
{L ,L ,L ,L ,L ,L }
  5 6 7 1 3 9
{L ,L ,L ,L ,L ,L }
  5 6 7 3 8 9
{L ,L ,L ,L ,L ,L }
  5 6 7 1 2 4
{L ,L ,L ,L ,L ,L }
  5 6 7 1 2 8
{L ,L ,L ,L ,L ,L }
  5 6 7 1 3 4
{L ,L ,L ,L ,L ,L }
  5 6 7 1 3 8

So for this example there can be no valid schedule with fewer than five processors.

Although we have developed a means of determining all of the irredundant coverings of LCMC's for a given job set, we as yet have no means of determining if a valid schedule exists among the irredundant coverings and at the same time we have no systematic technique for examining each of the coverings. In the following sections we will further define the structure of coverings of LCMC's of a given job set and develop a formal structure which contains every LCMC cover of a job set.

4.3 Essential Compatibles and Reduced Job Sets

In this section, we will begin our development of an algorithm designed to determine the minimal processor schedule for a given job set. As mentioned previously, our algorithm can be considered as consisting of two phases: the first phase of which will be developed in this section; the second phase will be developed in Section 4.6. The

initial phase will provide a means of partitioning the job set so that certain job assignments can be made without the necessity of considering all possible assignments of those jobs, yet, not excluding a minimal processor schedule.

In the previous section we illustrated how the minimal cover of LCMC's of a given job set establishes the lower bound for the minimal number of processors required for a valid schedule. In order to determine the minimal cover for a given job set, we first determined the collection of all of the maximal compatibles of the job set, expanded each of those maximal compatibles which were not load consistent into a set of load consistent maximal compatibles, and then determined all of the irredundant covers of LCMC's for the job set. The formation of LCMC's from a given MC that is not compatible, in many cases, is an unnecessary step as we will illustrate in this section. Furthermore, we will demonstrate that in some cases we can eliminate some of the possible job assignments prior to the creation of a load consistent collection and the generation of the irredundant covers for the job set. We will accomplish this by developing a technique for "reducing" the job set by assigning jobs to processors prior to forming all of the LCMC's of the job set, then removing those jobs from the collection of jobs to be scheduled, thereby reducing the job set. We will show that this "reduction" is dependent on the characteristics of the job set; and that it can be done without compromising our ability to determine a minimal processor schedule.

In previous discussions it was pointed out that certain LCMC's were essential to any cover of the job set by LCMC's. Appropriately, we will call such an LCMC an Essential LCMC (ELCMC). ELCMC's occur when

there exists within the collection of all LCMC's of a job set one or more LCMC which has a nonempty subset of jobs contained within no other LCMC of the job set. The ELCMC is the largest collection of jobs which are pairwise compatible and that contains that subset of jobs. Furthermore, every possible combination of jobs which contain that subset is within the ELCMC. We will refer to such a subset as an essential subset. An essential subset need not be exclusive to ELCMC's as it is possible for any cover to contain jobs which are elements of only one LCMC within that cover. In fact, every LCMC contained within any irredundant cover of LCMC's is essential to that cover and contains some subset of the job set not contained within any other LCMC of that cover. Not all job sets will contain ELCMC's, however, as this is dependent on the characteristic of the jobs in the set; but if there are ELCMC's in a job set each and every cover of LCMC's must contain each and every ELCMC. The ELCMC's and the associated essential subsets are therefore necessary for every cover; yet, there may exist LCMC's which are not ELCMC's but are "essential" to a given cover of the job set.

The technique we will use to partition the job set is based on the results of the following lemmas and the fact that our objective is to determine any minimal processor schedule for the job set. The concepts which will be developed enable us to "reduce" the job set to be scheduled. This "reduction" is dependent on the characteristics of the jobs in the set, and is in fact a scheduling of certain subsets of jobs before the job set is ordered for consideration of all possible job schedules.

Before we begin our development, we will define more precisely what we mean by "reducing" the job set. We will say that a successor

job set, K', of a job set K is any nonempty subset of K; K' is, of course, a job set in its own right. Therefore, a job set K is "reduced" when any subset of jobs, K'', is assigned to one or more processors and the successor job set, K', is formed with the jobs of K yet to be scheduled. The successor job set, K' for example, can be represented using set theoretic notation for the difference of two sets. That is, K'=K-A where K' is the set of jobs contained in K and not contained in the set of jobs A. In a similar manner, there exist predecessor job sets of any successor. The successor job set is determined by the jobs which have already been assigned to processors; there are many successor job sets associated with any given job set K so that arbitrary partitions of the job set may prevent the formation of a minimal processor schedule. For the construction of a minimal processor schedule of K, the job assignments performed in the "reduction" of the job set must be elements of the some minimal processor schedule in the collection of all minimal processor schedules for K. In the following lemma, we define an ordered way of partitioning the job set that insures any assignment made by this algorithm is one contained in the collection of minimal processor schedules.

Lemma 4.1 If for a job set $K=\{j_1, j_2, ..j_n\}$ with a collection of ELCMC's $\{EL_1, EL_2, ..EL_q\}$ with essential subsets $\{el_1, el_2, ..el_q\}$ respectively there exists a valid single processor schedule for a nonempty subset of the collection of ELCMC's, then the assignment of one of those ELCMC's, EL_i, to a processor will not increase the number of

processors required for any minimal processor schedule of the job set K. That is, if the minimal number of processors required for K is M, then the minimal number of processors required for the successor job set K-EL_i will be M-1.

Proof To prove this lemma we must show that there exists at least one M processor schedule of the job set K which contains the entire ELCMC, EL_i, on a single processor.

As stated previously, the ELCMC's contain every possible combination of jobs which can be assigned to the same processor as the essential subset of the ELCMC. If there exists a valid schedule for EL_i, then there is a single processor schedule for the essential subset, el_i, contained in EL_i. Moreover, any minimal processor schedule of K must contain the essential subset el_i on a single processor. In addition, since at least one processor will be required for the essential subset and only one ELCMC contains that subset, the assignment of the entire ELCMC to the same processor will result in no increase in the number of processors required for the entire job set unless the removal of a job from one of the other LCMC's of the job set would result in the change of a valid schedule of a subset of that LCMC into an impossible schedule assignment. Clearly if there is a valid schedule defined for a collection of jobs the removal of one job will not result in a conflict, hence this could only occur if the scheduling algorithm for a single processor were not optimal as defined in Chapter 3. Therefore, the assignment of the jobs in EL_i to a single processor using

the algorithm of Chapter 3 and the subsequent reduction of the job set will not increase the minimal number of processors required for the job set K. QED

The above lemma requires that the entire ELCMC have a valid single processor schedule for the ELCMC to be assigned to a processor. Suppose there exists a valid single processor schedule for more than one subset of the ELCMC but not for the entire ELCMC. Then any assignment of one or the other of the subsets to a processor does not consider the overall characteristics of the job set and the other LCMC's and may result in an increase in the number of processors required. For this reason such an assignment is not allowed or we can not be assured of a minimal processor schedule.

Also, in the above lemma the assignment of an ELCMC to a processor is limited to only one ELCMC with a valid single processor schedule. This is a result of the constraint imposed whereby each of the jobs in the job set may be assigned to one and only one processor. If two of the ELCMC's contain some jobs in common then an assignment of the two ELCMC's to processors would violate this constraint. The consequences of both of the above limitations will be examined in detail in the following discussion and it will be shown that neither will prevent a successful partitioning of the job set.

In the following lemma we will demonstrate that the essential LCMC's of the original job set also form part of the essential LCMC's of the successor job sets which contain those essential subsets, which have not been assigned to a processor.

Lemma 4.2 For a job set K with ELCMC's $\{EL_1, EL_2, ..EL_m\}$, the

ELCMC's of the successor job set K' formed from K-EL$_i$ include as a minimum the ELCMC's defined by {(EL$_1$ -EL$_i$), (EL$_2$ -EL$_i$),...(EL$_m$ -EL$_i$)}.

Proof Each of the ELCMC's of K contain an essential subset of jobs contained in no other LCMC. Therefore unless the entire essential subset, and in our case the entire ELCMC is assigned to a processor, the essential subset remains in the successor job set. QED

As a consequence of the above lemma, we know that given the essential LCMC's of the job set, K, we know the minimal collection of essential LCMC's for each successor job set, K', K'',... and so on of K. We will use this fact in the definition of an algorithm for finding a successor job set to K. But first, we will examine additional constraints on the formation of essential subsets and successor job sets.

Lemma 4.3 The maximal compatibles of any successor job set K' of the job set K are non-empty subsets of the maximal compatibles of K.

Proof The collection of all of the maximal compatibles of the job set K is determined by the characteristics of the jobs in the set. Each of the MC's represents the largest subsets of K that contain a certain subset of jobs which may have a valid single processor schedule. The collection of MC's contain every possible combination of those jobs. Therefore the collection of MC's also contain every possible combination of compatible jobs for any subset of K, K'; that is, any successor job set of K. While the compatiblility class formed by deleting jobs which have been assigned to a processor may not be maximal, there will remain

within the collection of all of the compatibility classes of the reduced maximal compatibles all of the maximal compatibles of the successor job set. QED

The above lemma does not say that all of the ELCMC's of the successor job set will be as defined above, but that the collection of ELCMC's of the successor job set will at least include those. There are as a matter of fact job sets in which there may occur ELCMC's other than those defined by the scheduling of an ELCMC.

Therefore, in the formation of a successor job set and the associated maximal compatibles, there may occur the creation of additional ELCMC's of the successor job set either from the reduction in the number of maximal compatibles, whereby an LCMC which was not essential for K is essential for K', or from the creation of an LCMC from a maximal compatible which was not load consistent for K by scheduling enough jobs so that the load factor becomes less than unity.

The algorithm we will now define accepts as input a given job set K and all of its maximal compatibles and determines for each essential LCMC of K a valid schedule if it exists and reduce the job set accordingly. The resulting job set may in fact be null, all of the jobs scheduled, or a collection of compatibles some of which may not be load consistent.

Algorithm 4.1

Step 1 Form all of the maximal compatibles of the job set $K=\{j_1, j_2, \ldots j_n\}$ and form the set of all ELCMC's of K, $\{EL_1, EL_2, \ldots, EL_s\}$.

Step 2 Let $K^* = K$ and $EL_i^* = EL_i$ for all $i=[0,1,..s]$.

Step 3 Test for ELCMC's for the job set K^*. If $s = 0$ then there exist no ELCMC's for the job set K^*. Stop. This phase of the multiprocessor scheduling algorithm is complete. If s is not zero then there exists at least one ELCMC of the job set K^*.

Step 4 Order the ELCMC's EL_i^* by nondecreasing cardinality. Set Index = 0; set 1 = 0.

Step 5 Let Index= Index + 1

Step 6 Test EL^*_{Index} for a valid single processor schedule. If there exists a valid single processor schedule, go to step 7. If not, then store the results of the test in the lists of feasible schedules, F and the list of infeasible schedules , I. If Index = s, then go to Step 10; the entire list of ELCMC's of K^* has been evaluated and the last element has no valid single processor schedule. If Index = s then go to Step 5.

Step 7 Schedule EL^*_{Index} on a processor. That is, let 1=1+1 and $P = \{EL^*_1, \ldots EL^*_{Index}\}$.

Step 8 Form the minimal collection of ELCMC's of the successor job set $K' = K - EL^*_{Index}$; reorder the list of ELCMC's; let s = s-1.

Step 9 If s=0 then the entire list of ELCMC's of the job set

K^* has been tested and every ELCMC, or reduced ELCMC, has a valid single processor schedule. If s is not equal to zero, set Index =0, begin testing at the start of the list, and go to Step 5.

Step 10 Form the reduced job set $K'' = K-\{j_m : j_m$ is an element of some $P_1\}$. If K''is empty, all of the jobs in K have been scheduled; this phase of the multiprocessor algorithm has constructed a minimal multiprocessor schedule of K - Stop.

Step 11 Form all of the maximal compatibles of K''. Find all of the ELCMC's of K'', $\{EL_1,..EL_s\}$; and let $K^* = K''$. Go to Step 3.

The algorithm presented above is illustrated in Figure 4-1.

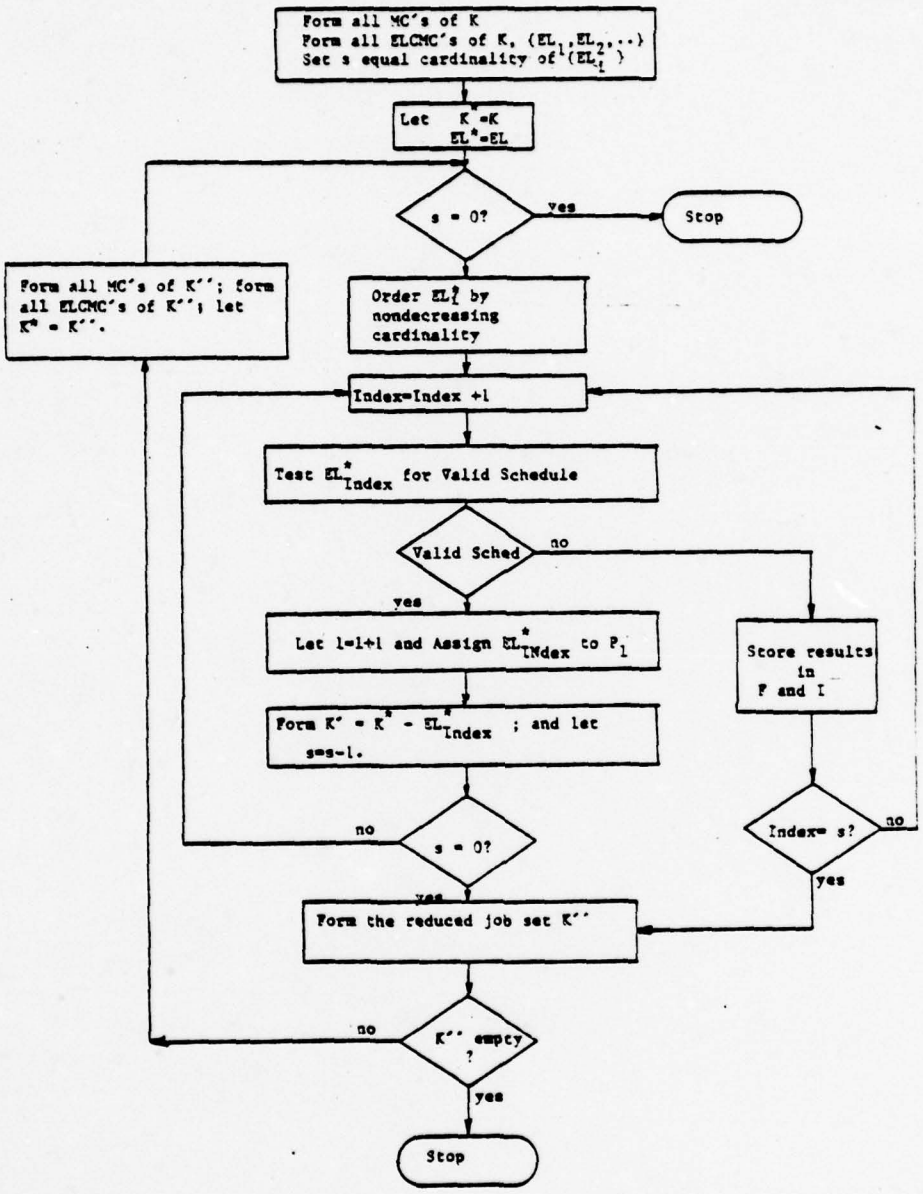


Figure 4-1 Phase I - Minimal Processor Scheduling Algorithm

At this point we will demonstrate the use of this phase of the

multiprocessor scheduling algortihm by analyzing its application to a specific example job set. Consider the job set used in the example of Section 2.5.

Job	Period	Execution Time
1	1	0.5
2	1	0.4
3	2	1.6
4	2	1.4
5	3	0.5
6	4	0.5
7	6	0.8
8	8	1.5
9	8	0.6
10	9	0.5

As was illustrated in Section 2.5, there are seven maximal compatibles of this particular job set as follows:

$A = \{j_1, j_2, j_5, j_6, j_{10}\}$

$B = \{j_5, j_6, j_7, j_{10}\}$

$C = \{j_6, j_7, j_9\}$

$D = \{j_2, j_6, j_9\}$

$E = \{j_6, j_8, j_9\}$

$F = \{j_4, j_6, j_9\}$

$G = \{j_3\}$

First we will determine for this job set those maximal

compatibles which are essential. The table listed below represents the maximal compatible cover table for the job set:

	1	2	3	4	5	6	7	8	9	10
A	1	1	0	0	1	1	0	0	0	1
B	0	0	0	0	1	1	1	0	0	1
C	0	0	0	0	0	1	1	0	1	0
D	0	1	0	0	0	1	0	0	1	0
E	0	0	0	0	0	1	0	1	1	0
F	0	0	0	1	0	1	0	0	1	0
G	0	0	1	0	0	0	0	0	0	0

The essential MC's are easily determined by finding those columns of the cover table that contain a single one, where a 1 in column k and row I indicate job j_k is contained in MC, I. The MC that contains the job represented by that table entry is the only MC to contain that job and is therefore an essential MC. Going from left to right the essential MC's are:

$A = \{j_1, j_2, j_5, j_6, j_{10}\}$

$G = \{j_3\}$

$F = \{j_4, j_6, j_9\}$

$E = \{j_6, j_8, j_9\}$

Of the above listed MC's of the example job set only A is not load consistent. Hence, each of these essential LCMC's could in fact represent a valid single processor and are therefore ELCMC's of the job set. For ease of reference we will not rename these ELCMC's prior to

using the algorithm. But since the number of ELCMC's is small for this example, it will be easy to relate the example to the algorithm.

We begin the process by forming the list of ELCMC's of the job set. We will order the ELCMC's by nondecreasing cardinality. That is G, E, F.

We test the first ELCMC, G, in the list for a valid single processor schedule. Clearly there exists such a schedule, hence we create a processor, P_1, with the schedule $P_1 = \{j_3\}$.

There are no jobs in P_1 that are also contained in any of the other ELCMC's of the job set; so those ELCMC's that remain are not changed.

Next we test the ELCMC, E, for a valid single processor schedule. Since it is not relevant to the purposes of this example, (i.e., to illustrate the reduction of a job set using Algorithm 4.1), we will not illustrate the complete operation of the single processor scheduling algorithm, but note the following:

$$\gcd(T_6, T_8, T_9) = \gcd(4, 8, 8) = 4$$

and

$E_6 + E_8 + E_9 = 0.5 + 1.5 + 0.6 = 2.6 < \gcd(T_6, T_8, T_9)$ so that there exists a valid schedule for E on a single processor. In fact, one such schedule is defined for the processor $P_2 = \{j_6, j_8(0.5), j_9(2.0)\}$.

In this instance there are jobs in the remaining ELCMC, F, that are in the schedule of processor P_2. They are j_6 and j_9. Hence these

jobs are removed from F and a "new" ELCMC, F′, of the successor job set is formed.

Again, there is a valid schedule for the ELCMC, F′. A processor and an associated schedule $P_3 = \{j_4\}$ are created.

At this point all of the ELCMC′s have been eliminated so that there exists a successor job set, different from the original job set, which we must now specify and consider for a minimal processor schedule.

The successor job set that is formed after the scheduling of j_3, j_4, j_6, j_8, and j_9 is $K' = \{j_1, j_2, j_5, j_7, j_{10}\}$. The compatability classes that are formed by deleting the jobs which were scheduled from the maximal compatibles of K results in the following collections of jobs.

$A' = \{J_1, j_2, j_5, j_{10}\}$

$B' = \{j_5, j_7, j_{10}\}$

$C' = \{j_7\}$

$D' = \{j_2\}$

Of these only A′ and B′ are maximal compatibles of the successor job set, K′. The essential MC′s are, in this case, all of the MC′s of the job set, but again the maximal compatible A′, is not load consistent.

If there had been no more LCMC′s formed for the successor job set or if none of the LCMC′s were essential then the reduction of the job set via this algorithm would have been completed and the remaining scheduling tasks would require the next phase of the minimal processor scheduling algorithm.

Notice, in addition to the original essential MC which was not load consistent, the successor job set, K′, contains an essential MC which resulted from the elimination of some of the jobs from K. This MC is in fact load consistent and therefore an ELCMC.

Again, we order the ELCMC′s by nondecreasing cardinality and employ the algorithm to determine a successor job set. As was the case with the ELCMC, E, illustrated previously, the sum of the execution times of the jobs in B′ do not exceed the greatest common divisor of the job periods so that a valid single proceesor schedule exists. One such schedule is $P_4 = \{j_5, j_7 (0.5), j_{10} (1.3)\}$.

Since this was the only ELCMC of K′, the successor job set, K′′, must now be formed. (If there were no valid schedule for B′, then this phase of the algorithm would be completed.) The only remaining compatability class, A′, is reduced by eliminating those jobs assigned to P_4 and the new compatability class $A'' = \{j_1, j_2\}$ is formed. Notice this is now an ELCMC of the successor job set, K′′, and is the only MC remaining. Its assignment to $P_5 = \{j_1, j_2 (0.5)\}$ represents final determination of a minimal processor schedule for the job set K.

The schedule that we have just constructed is as follows:

$P_1 = \{j_3\}$

$P_2 = \{j_6, j_8 (0.5), j_9 (2.0)\}$

$P_3 = \{j_4\}$

$P_4 = \{j_5, j_7 (0.5), j_{10} (1.3)\}$

$$P = \{j_5, j_1 (0.5)\}$$
$$2$$

We should point out prior to completing this section of the chapter certain relationships between the results of this example and previous results. First, if the reader refers back to Section 4.2 and the illustration of the determination of the irredundant covers of LCMC's for the job set of this example, he will note that the minimal cover has five LCMC's. That is, the minimal number of processors required for this job set cannot be less than five. Furthermore, each of the subsets of the job set K which were assigned to specific processors are subsets of one of the LCMC's contained in the cover.

Also, in the example used above, both instances in which ELCMC's of a successor job set can be created from non ELCMC's of a predecessor job set are shown. The first reduction of the job set resulted in the formation of an ELCMC from an MC which was not essential for the original job set. Finally, the MC which was not load consistent for the first two successor job sets eventually formed an ELCMC of the final successor job set which was considered.

In this section we have formulated the initial phase of an algorithm for determining a minimal processor schedule of a given job set. This initial phase, while not essential to the determination of a minimal processor schedule, provides a basis for reducing the computation required for some job sets in defining a minimal processor schedule. It accomplishes this by providing a means of partitioning the set of jobs to be scheduled and assigning certain subsets of jobs to specified processors. While this phase of the scheduling algorithm does eliminate certain possible schedules from consideration, it does not preclude the determination of a minimal processor schedule.

In the next section we will continue our specification of a general algorithm for minimal processor scheduling by examining partitions of the job set that may have a valid single processor schedule but are not dependent on the existence of essential subsets of the job set.

4.4 Set Systems of LCMC

In this section we will develop the formalism necessary to limit the collections of LCMC's to those that may contain a valid multiprocessor schedule. We will also define a lattice structure for the covers of LCMC's that we are able to construct systematicaly given all of the irredundant LCMC covers.

A set system S of a set K is a collection of subsets $\{B_i\}$ of K such that:

(1) Every element of K is contained in at least one block B_i of S.

(2) $B_i \subseteq B_j$ implies i=j. That is, no block of the set system is a proper subset of any other block of the set system (Har).

It follows from the definition of a set system that all partitions of a set are also set systems of that set with the additional distinction that the intersection of each pair of blocks B_i and B_j is null for all i not equal to j.

For a finite set of elements, condition (2) above guarantees that there will be only a finite number (though perhaps a large number) of set systems for that set. This fact enables us to order the systems of a given set (Har).

We define an ordering of two set systems S_1 and S_2 of a finite job set K as follows:

$S_1 \geq S_2$ if and only if for each block B_i^2 of S_2, there exists a block B_j^1 of S_1 such that $B_i^2 \subseteq B_j^1$.

As an example, for a set {1,2,3,4,5}, two of the possible set systems are {(1,2,3);(3,4,5);(2,3,4)} and {(1,2);(3,4);(5)}; and

{(1,2,3);(3,4,5);(2,3,4)}$\geq${(1,2);(3,4);(5)}.

The set system {(1,2,3);(3,4,5)} is also less than the first set system shown above, but is larger than the second set system.

In order to define a formal structure for collections of LCMC's we will first examine the structure of set systems in general and then relate this structure to an acceptable structure for the problem of multiprocessor scheduling of periodic tasks.

The maximum of an arbitrary collection of subsets $\{A_i\}$ of a finite set A is defined as follows:

$Max\{A_i\}=\{B \subseteq A: A_i = B$ for some i and $A_j \supseteq B$, implies $A_j = B\}$. In words, $Max\{A_i\}$ is the collection of sets A_i not properly contained in any other set A_j of the collection.

For example, Max{ (1,2);(1,2,4);(3,5);(3)} = {(1,2,4);(3,5)}.

It has been shown that for a finite set the set of all set systems of that set form a distributive lattice under the ordering given above and the lattice operations defined as follows:

For set systems S_1 and S_2 of a set A,

$S_1 + S_2 = \text{Max}\{B : B \in S_1 \text{ or } B \in S_2\}$ and

$S_1 * S_2 = \text{Max}\{B_i \cap B_j : B_i \in S_1 \text{ and } B_j \in S_2\}$ (Har).

As a consequence of our previous development, we know that the LCMC's of a job set contain the combinations of jobs which may be contained in a valid schedule. We are therefore interested in the combinations of LCMC's that cover the job set, that is the set systems of LCMC's of the job set. The set systems of LCMC's of a given job set are easily formed from the irredundant covers of LCMC's of the job set. Since each cover of LCMC's must contain at least one of the irredundant covers, all of the covers are formed by successively forming all of the combinations of LCMC's that cover the job set by forming all combinations of LCMC's for each of the irredundant covers and then eliminating the duplicate covers.

The set systems formed from LCMC's of a job set K, are of course elements of the collection of set systems of the job set and therefore, elements of the lattice of set systems of K. But, due to the property of maximal compatibles which cover a job set, the set systems of LCMC's of a job set do not, in general, form a lattice. It is inherent of course from the definition of a set system that any set system of LCMC covers the job set. Yet, consider the meet of the two set systems of LCMC's shown below:

The meet of the set systems of LCMC's {(1,2,3);(2,3,4);(3,5,6)} and {(1,2,6);(2,3,5);(4,6)} form {(1,2);(2,3);(4);(3,5);(6)} which is not a collection of LCMC's.

Although the set systems of LCMC's do not form a lattice, there does exist a lattice structure for these set systems.

A partial ordered set $S = \{S_1, S_2, \ldots S_p\}$ is an upper (lower) semilattice if and only if S is closed under $S_i + S_q$ ($S_i * S_q$) (Bir).

Thus for any two elements of an upper semilattice, there must be a least upper bound also contained in the semilattice; but, unlike a lattice, the greatest lower bound of any two elements need not be in the upper semilattice. Each upper semilattice contains a unique identity element, which for set systems of LCMC's is the collection of all LCMC's of the job set. In a similar manner, for any two elements of a lower semilattice there must be a greatest lower bound also contained in the semilattice. A lower semilattice contains a unique universal element. We will now show that the set systems of LCMC's of a job set form an upper semilattice.

Theorem 4.2 The collection of set systems of LCMC's of a job set K with the ordering and the join (+) operation defined above, forms an upper semilattice with a common identity element equal to the collection of all of the LCMC's of K.

Proof Suppose there exists two set systems of LCMC's S_i and S_p of the job set K. The join of S_i and S_p is $S_i + S_p = \text{Max}\{B: B \text{ in } S_i \text{ or } B \text{ in } S_p\}$.

For set systems of LCMC's, the join operation is equivalent to the union of the blocks of the individual LCMC's of the set systems. Thus $S_i + S_p$ is a set system of LCMC's that consists of the collection of

all of the LCMC's in S_i and the LCMC's in S_p. QED

As an example of the formation of the semilattice of set systems of LCMC's consider the example job set of Section 4.2. We will assume that there is no reduction possible for this set and will form the semilattice from the collection of all of the irredundant covers of LCMC's.

For ease of discussion, we will refer to the level of the semilattice of set systems of LCMC's, where the level, l, is the collection of all of the set systems with l LCMC's.

We form the semilattice a level at a time begining with the minimal set system level, in this case the single five element set system, and end with the identity element of the semilattice, which is the collection of all of the LCMC's of the job set. For this example, there are nine LCMC's so that the semilattice contains five levels - five thru nine.

Any minimal cover is always irredundant. The collection of the minimal covers form the first level.

The next level is composed of all irredundant covers with one more than the minimal covers and the all combinations of the minimal covers with the LCMC's not contained in each of the minimal covers.

The levels are formed successively by adding irredundant covers as they occur and the join of existing elements of the semilattice.

For the example job set,
Level 5:

$$\{L_2, L_5, L_6, L_7, L_9\}$$

Level 6:

The redundant set systems:

{L ,L ,L ,L ,L ,L }
  1 2 5 6 7 9
{L ,L ,L ,L ,L ,L }
  2 3 5 6 7 9
{L ,L ,L ,L ,L ,L }
  2 4 5 6 7 9
{L ,L ,L ,L ,L ,L }
  2 5 6 7 8 9

The irredundant set systems:

{L ,L ,L ,L ,L ,L }
  1 2 4 5 6 7
{L ,L ,L ,L ,L ,L }
  1 2 5 6 7 8
{L ,L ,L ,L ,L ,L }
  1 3 4 5 6 7
{L ,L ,L ,L ,L ,L }
  1 3 5 6 7 8
{L ,L ,L ,L ,L ,L }
  1 3 5 6 7 9
{L ,L ,L ,L ,L ,L }
  3 5 6 7 8 9

Level 7:

There are no irredundants:

{L ,L ,L ,L ,L ,L ,L }
  1 2 3 5 6 7 9
{L ,L ,L ,L ,L ,L ,L }
  1 2 4 5 6 7 9
{L ,L ,L ,L ,L ,L ,L }
  1 2 5 6 7 8 9
{L ,L ,L ,L ,L ,L ,L }
  2 3 4 5 6 7 9
{L ,L ,L ,L ,L ,L ,L }
  2 3 5 6 7 8 9
{L ,L ,L ,L ,L ,L ,L }
  2 4 5 6 7 8 9
{L ,L ,L ,L ,L ,L ,L }
  1 2 3 4 5 6 7
{L ,L ,L ,L ,L ,L ,L }
  1 2 4 5 6 7 8
{L ,L ,L ,L ,L ,L ,L }
  1 2 3 5 6 7 8
{L ,L ,L ,L ,L ,L ,L }
  1 3 4 5 6 7 8
{L ,L ,L ,L ,L ,L ,L }
  1 3 4 5 6 7 9
{L ,L ,L ,L ,L ,L ,L }
  1 3 5 6 7 8 9
{L ,L ,L ,L ,L ,L ,L }
  3 4 5 6 7 8 9

Level 8:

All of the eight LCMC subsets of the nine LCMC's of the job set.

Level 9:

$$\{L_1, L_2, L_3, L_4, L_5, L_6, L_7, L_8, L_9\}$$

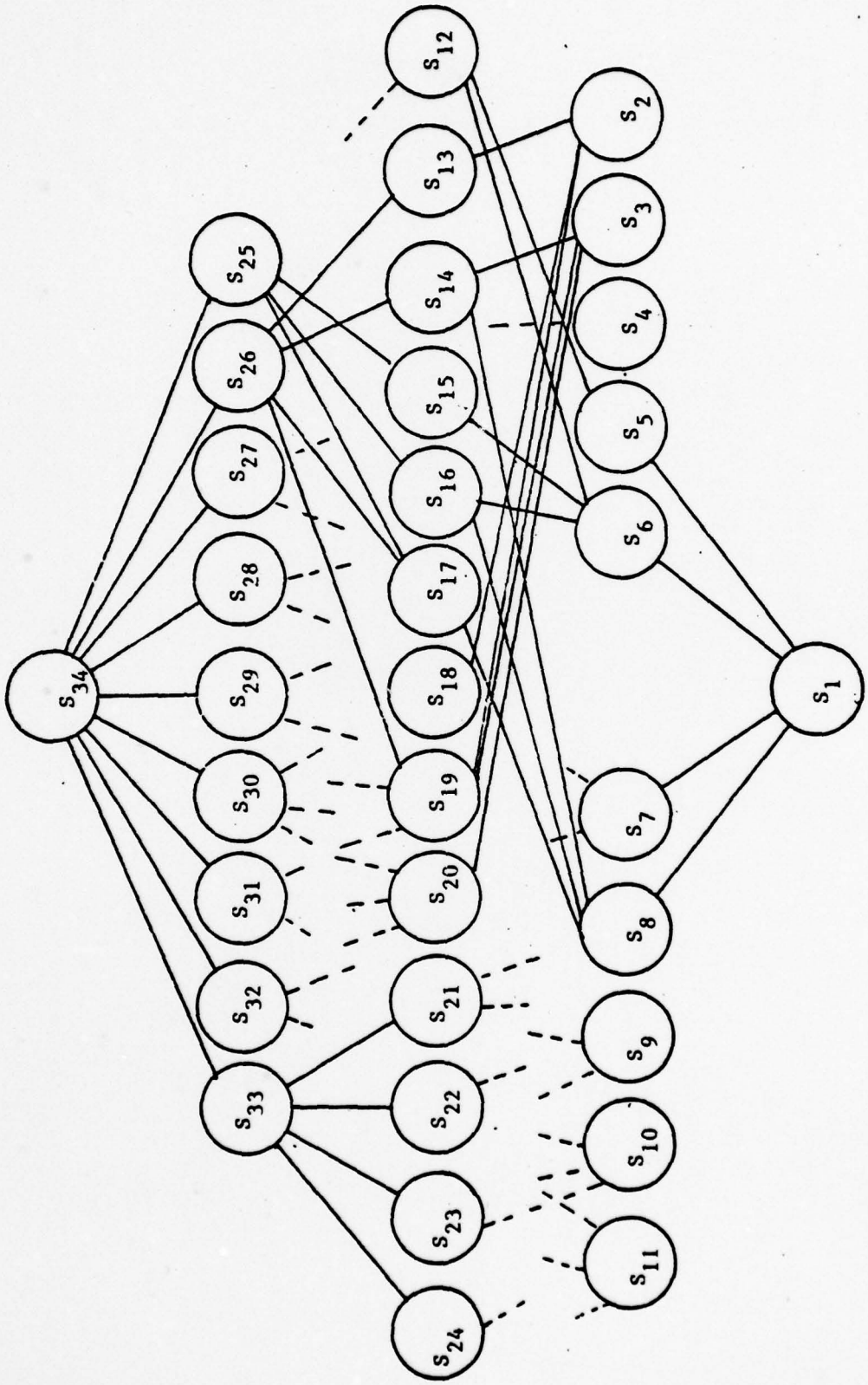


Figure 4-2 Semilattice of Set Systems of LCMC's

Since every valid multiprocessor schedule of a job set is a partition of that job set such that there exists a valid uniprocessor schedule for each block of the partition, we will necessarily have to determine the partitions of the job set that are specified by some set system of LCMC's. We will see in the next section that each set system in the semilattice specifies a set of partitions each element of which may represent a valid multiprocessor schedule for the job set. Our objective will be to define a technique which limits the partitions that must be formed.

4.5 Admissible Partitions

The semilattice of set systems of LCMC's defined above contains all of the possible combinations of LCMC's which may contain a valid schedule. But, as stated in the formal definition of the scheduling problem in Chapter 2, a valid schedule of a job set is a partition of the set. Yet, in general, set systems of LCMC's are not partitions. In this section, we will formulate the concepts and techniques that will enable us to examine the different partitions of the job set which may represent a valid schedule.

While it is conceptually not difficult to form every partition with a given number of blocks of a job set K and to compare each partition with the set systems of LCMC's of K, such a task could require a significant amount of computer resources. In addition, many of the partitions formed would be discarded as not contained in a set system of LCMC's and therefore not representative of a possible valid schedule. For example, there are 43525 five-block partitions of a job set with ten jobs, but for the job set of the previous example, it can be shown that

only twelve of these partitions can possibly represent a valid schedule for the job set in question. That is, only twelve partitions are contained in the minimal covering of LCMC's.

In this section we will formulate a procedure which will enable us to generate only those partitions of the job set that could possibly represent a valid schedule of the job set. Also, we will show that there exists a lattice structure of these partitions and that every possible valid schedule of the job set is an element of this lattice structure.

A partition $P=\{B_1,\ldots,B_m\}$ of a job set K is said to be an admissible partition (AP) if every block, B_i, of P is contained within some LCMC of K. That is, a partition is admissible if it consists of LCC's of the job set K.

From our previous development of LCC's and coverings of LCMC's, it is clear that only admissible partitions can possibly contain a valid schedule of the job set. We now define an order relation on the partitions of a set.

For partitions P_1 and P_2 on a set K, the partition P_1 is said to be smaller than P_2 denoted

$$P_1 \leq P_2$$

iff each block of P_1 is contained in some block of P_2.

We can say that the partition P_1 is a refinement of the

partition P_2. It is inherent in the definition of the ordering relation that any refinement of a partition contains more blocks that the partition itself. (Note: In the same manner, an arbitrary set system S_1 is a refinement of a set system S_2 if $S_1 \leq S_2$.)

In order to describe the binary operations on partitions, we must introduce a set-theoretic property regarding the intersection of sets. Two sets A and B are said to be connected if the intersection of A and B is not null. In addition if A and B are two subsets from a collection C of subsets of the set K, then we say that A and B are chain-connected in C if there exists a finite sequence $<A_i > 1 \leq i \leq j>$ of subsets in C such that $A = A_1$, $A_j = B$, and A_i is connected to A_{i+1} $(1 \leq i \leq j-1)$ (Har). Based on these definitions we define two binary operations on partitions as follows:

For paritions $P_i = \{A_1, \ldots A_m\}$ and $P_j = \{B_1, \ldots B_n\}$, $P_i * P_j$ is the partition on K such that $P_i * P_j = \{A_q \cap B_t$; A_q is an element of P_i, B_t is an element of P_j, and A_q is connected to $B_t\}$.

$P_i + P_j$ is the partition of K such that $P_i + P_j = \{A_k = \cup A'$; A' is chain-connected to A_k in $P_i \cup P_j$ $(k = 1, .., m)\}$.

$P_i * P_j$ = greatest lower bound (P_i, P_j) and $P_i + P_j$ = least upper bound (P_i, P_j) (Pra).

The set of all partitions of a finite set forms a lattice under the partial order defined above with these binary operations (Pra). But, as was true for the set systems of the LCMC's, the collection of AP's of a job set forms a semilattice. In this case, as we will prove in the next lemma, the collection of all AP's of a given job set forms a lower semilattice.

Lemma 4.4 The collection of all AP's of a given job set forms a lower semilattice with a zero element of a single job in each block of the partition.

Proof We must show that the collection of AP's is closed under the operation *. That is, for two arbitrary AP's, P_i and P_j, their meet is also an AP of the job set K.

Clearly, the zero element of the semilattice as defined above is an admissible partition.

From the definition of AP's of a job set and the binary operation *, the result of the meet of two AP's is a partition in which every block is a subset of some block of the paritions P_i and P_j. Hence, every block of $P_i * P_j$ is also an LCC of K and the partition is therefore an AP. Therefore the AP's are closed under the meet operation. Clearly, given the definition of AP's, they would not, in general, be closed under the join operation. QED

Given that all of the possible partitions of the job set for which a valid schedule may exist is contained in the semilattice of AP's for the job set, its construction is a key element in the determination of a minimal processor schedule. We will define a technique for

constructing the lattice structure by noting that there exists for each of the LCMC set systems a lower semilattice of AP's. Further, since each LCMC set system of the job set contains at least one irredundant set system and the semilattice of AP's must contain the semilattice of any refinement of that set system, we are able to construct the semilattice of the admissible partitions of the job set from the admissible partitions of the irredundant LCMC set systems.

In the example to follow, we will show the determination of the AP's of an irredundant set system. We will then expand upon this technique to determine the semilattice of AP's for the that irredundant set system. We will construct this semilattice a level at a time (where the level is determined by the number of blocks in the AP) and only form partitions of increased cardinality after every partition which may represent a valid schedule and contains fewer blocks has been constructed and tested. In the next section, we will further expand this concept by defining an algorithm for constructing the semilattice of AP's for the entire job set (along with the determination of a minimal processor schedule). The semilattice for the job set will also be constructed a level at a time.

The method of construction of the semilattice a level at a time is a primary aspect of the minimal processor scheduling algorithm. This method assures that the partitions of the job set are considered in the order of nondecreasinng cardinality. Hence, once a partition is found which has a valid single processor schedule for every block, the schedule is a minimal processor schedule.

Consider the minimal LCMC set system of the example job set of Section 4.2. The set system contained five LCMC's. The AP's we will

form will consist of five blocks and are all of the five block paritions of that LCMC set system. Every refinement of those five block partitions can then be formed from these partitions by creating the six then seven, and so on, block refinements.

The LCMC set system $\{L_2, L_5, L_6, L_7, L_9\}$ is a minimal set system. Since it is irredundant, each and every LCMC is essential to this particular set system. Thus, there is at least one job within each LCMC which must occur within any LCC formed from that LCMC. The LCMC's of the set system above are as follows:

$L_2 = (j_2, j_5, j_6, j_7, j_{10})$

$L_5 = (j_5, j_6, j_8, j_9)$

$L_6 = (j_6, j_4, j_6, j_9)$

$L_7 = (j_7, j_3)$

$L_9 = (j_9, j_1, j_2, j_{10})$ with essential subsets (j_5, j_7), (j_8), (j_4), (j_3), and (j_1, j_2) respectively.

In order to form the AP's with the minimal number of blocks in the partitions, we must specify for each LCMC every LCC which might form a block within a partition. Clearly, those jobs which are essential to a given LCMC must be contained within every LCC formed from that LCMC.

For the set system of our example, the LCC's that might represent a block of a partition are

$B_1 = (j_1, j_2)$

$B_2 = (j_2, j_1, j_2, j_{10})$

$B_3 = (j_4)$

$B_4 = (j_4, j_6)$

$B_5 = (j_4, j_9)$

$B_6 = (j_4, j_6, j_9)$

$B_7 = (j_8)$

$B_8 = (j_6, j_8)$

$B_9 = (j_8, j_9)$

$B_{10} = (j_6, j_8, j_9)$

$B_{11} = (j_5, j_7)$

$B_{12} = (j_5, j_6, j_7)$

$B_{13} = (j_5, j_7, j_{10})$

$B_{14} = (j_5, j_6, j_7, j_{10})$

which are just the LCMC subsets that contain the essential subsets.

The determination of the AP's of five blocks is made by a procedure similar to that used in Section 4.2 to determine all of the irredundant set systems once the cover table has been reduced. The one difference between the two solutions is that the determination of the AP's is made with a boolean function of products of <u>exclusive or's</u>, which we will denote by $\pm$, of LCC's. That is, for this example, the set

of all minimal block AP's is given by the solutions to the boolean function

$$f(AP)=(B_1 \pm B_2)(B_3 \pm B_4 \pm B_5 \pm B_6)(B_{11} \pm B_{12} \pm B_{13} \pm B_{14})(B_4 \pm B_6 \pm B_8 \pm B_{10} \pm B_{12} \pm B_{14})(B_7 \pm B_8 \pm B_9 \pm B_{10})(B_5 \pm B_6 \pm B_9 \pm B_{10})(B_2 \pm B_{13} \pm B_{14})$$

This function has been reduced to eliminate the single job j_3 and the duplicate elements which result when there is an essential subset with more than one job. The solution of this equation yields twelve five block partitions of the set system above. For shorter notation we will represent these partitions by blocks of job numbers where N represents j_N.

(1,2)(3)(4)(5,6,7,10)(8,9)

(1,2)(3)(4,9)(8)(5,6,7,10)

(1,2)(3)(4,6,9)(8)(5,7,10)

(1,2,10)(3)(4,6,9)(8)(5,7)

(1,2)(3)(4)(5,7,10)(6,8,9)

(1,2,10)(3)(4)(5,7)(6,8,9)

(1,2,10)(3)(4,6)(5,7)(8,9)

(1,2,10)(3)(4,9)(5,6,7)(8)

(1,2,10)(3)(4,9)(5,7)(6,8)

(1,2,10)(3)(4)(5,6,7)(8,9)

(1,2)(3)(4,6)(5,7,10)(8,9)

(1,2)(3)(4,9)(5,7,10)(6,8)

These are all of the possible five block partitions of the set system above and, for this example, the job set.

By using the algorithm defined above, the minimal AP's for each of the irredundant set systems of the job set can be constructed. The construction of the semilattice of AP's for all of the set systems which contain the minimal set system is as follows:

The semilattice of AP's defined by the minimal AP's of the set system determines every possible partition that can be formed from this irredundant set system. In addition to these partitions there are AP's which can be formed from the redundant set systems in the semilattice of set systems of LCMC's.

Consider the redundant set system formed when the minimal set system is combined with the LCMC $L_3 = (j_3, j_6, j_7, j_9)$. The only five-block partitions of this cover are those we determined above for the irredundant set system. The six-block partitions include those formed from the five-block partitions and the six-block partitions formed by creating a block of the partition from the subsets of L_3 and eliminating those jobs from the five-block partitions formed above. The actual partitions which need to be formed from the minimal block partitions to define the next level of the semilattice, without considering the LCMC L_3, does not include all of the partitions it is possible to form from the minimal AP's. For example, there is no advantage in forming two one-job blocks from a single block with two jobs, e.g. (1,2), since there exists a single processor schedule for that pair of jobs. Likewise, there is no advantage in creating a partition of any LCC for which a valid single processor schedule is know to exist. Consider the job set of this example, we have shown that there exists a valid

schedule for j_6, j_8, and j_9 on a single processor (see Section 4.3) hence there is no advantage to further paritition this LCC.

The six-block AP's of interest which are formed from the twelve AP's with five blocks include all refinements of those AP's that are created without partitioning a block with a valid single processor schedule.

There are, in addition, six-block AP's that result from the set system formed with $L = \{j_3, j_6, j_7, j_9\}$.

The only different AP's that are formed from the set system created when L_3 is combined with the irredundant set system are those that result from the subsets of L_3 which are not contained within any LCC of the irredundant set system. All other AP's will be formed, either explicitly or implicitly, from the collection of AP's of the irredundant set system.

For L_3, only $\{j_7, j_9\}$ and $\{j_6, j_7, j_9\}$ represent subsets of the job set that are not already LCC's of the irredundant set system. All of the six-block partitions formed for this combination of LCMC's result by including these subsets as blocks of the AP's and eliminating these jobs from the blocks of the minimal AP's. Thus, the AP {(1,2)(3)(4)(5,6,7,10)(8,9)} provides a basis for two AP's in addition to those formed by partitioning the blocks above; they are {(7,9)(1,2)(3)(4)(5,6,10)(8)} and {(6,7,9)(1,2)(3)(4)(5,10)(8)}. Each of the other minimal AP's also forms the basis of other six-block partitions with the subsets of L_3.

There are also six-block partitions formed when the irredundant set system is combined with the other LCMC's of the job set such as L_1, L_4, and L_8 to create redundant set systems. The complete set of all six-block partitions of the job set defined by this irredundant set system is therefore formed by finding first the six-block partitions of the irredundant set system and then forming all of the six-block partitions of the redundant set systems defined by all six LCMC set system which contain this irredundant.

Extending this concept further, when the set system at the next level is formed by combining all pairs of the LCMC's L_1, L_3, L_4, and L_8 (L_3 and L_1 for example) with the minimal set system, the seven-block partitions are determined by the the six-block partitions defined previously, which include the six-block partitions from L_1 and L_3, and the creation of the seven block partitons consisting of a single block from each of the LCMC's of the pair (e.g., L_3 and L_1) respectively and the elimination of these jobs from the blocks of the minimal AP's.

In the case of the combination of L_1 and L_3 with the minimal set system, there is more than one irredundant set system contained within the cover. This set system includes also the irredundant {L_1, L_3, L_5, L_6, L_7, L_9}. The AP's of any set system formed from this combination will contain exactly the same AP's and only one of the irredundant set systems need be considered further.

The lower semilattice formed by the minimal set system is

created a level at a time by examining the upper semilattice of set systems that is defined by the irredundant set system until all possible combinations of LCMC's have been created, i.e., when the identity element of the LCMC semilattice is partitioned. After that point every possible AP of the job set is a refinement of the preceeding level of the lower semilattice of AP's.

Each of the irredundant set systems of a given job set defines a lower semilattice of AP's which has a common identity element with every other semilattice of AP's for the job set. The collection of all of these semilattices defines all of the AP's of the job set.

In the next section, we will define an algorithm for constructing the lower semilattice of AP's of the job set a level at a time. We will in addition examine each of the AP's as they are created to determine if they represent a valid schedule of the job set and therefore construct a minimal processor schedule in the process.

4.6 Minimal Processor Scheduling Algorithm

We have now completed the development of the basic concepts necessary to define an algorithm to construct a minimal processor schedule for a given set of periodic jobs. The optimality of the algorithm is based on three properties of the previous developments. These are:

- The characterization defined in Chapter 2, (the greatest common divisor of job periods and the total load factor of a collection of jobs), defines subsets (LCMC's) of the job set which may be scheduled on a single processor. Each and every possible subset of jobs that may be assigned to a single processor is a subset (LCC) of one of the LCMC's.

- All of the irredundant set systems, which includes all minimal set systems, of LCMC's for a given job set can be determined. And, the semilattice of set systems formed from the irredundant set systems orders all of the possible collections of jobs and defines a semilattice of Admissible Partitions (AP's) of the job set. The semilattice of AP's defines every possible partition of the job set and can be formed a level at a time so that a level containing partitions with q blocks is formed only after every partition with q-1 or fewer blocks has been constructed.

- The algorithm defined in Chapter 3 is optimal in the sense that for a given LCC of jobs it can determine if there exists a valid schedule for that LCC on a single processor.

The algorithm we will now define will construct a minimal processor schedule for a given job set, or successor job set. This algorithm may be considered as the second phase of the overall minimal processor algorithm, where the first phase of the algorithm (Algorithm 4.1) involves the reduction of the job set. But, this phase of the algorithm will determine a minimal processor schedule whether or not there has been an attempt to reduce the job set. It does not require that the initial phase precede it since this algorithm will construct a minimal processor schedule for whatever job set is input.

Algorithm 4.2

Step 1 Form all of the LCMC's of the job set K. The total number of LCMC's determines the highest level of the semilattice of LCMC set systems. Set LMAX equal to the cardinality of the set of LCMC's, #LCMC.

Step 2 Form all of the irredundant set systems of the job set. Order the set systems by nondecreasing cardinality $\{S_1,\ldots,S_q\}$. The lowest level of the semilattice is $S_1^{\#}$. Set the current level of the semilattice (CL) equal to $S_1^{\#}$. Set the index of the irredundant set systems, i, equal to zero.

Step 3 Increment the index of irredundant set systems (i=i+1).

Step 4 Form all of the Admissible Partitions of the set system S_i, $AP(S_i)=\{ap_{i1},\ldots,ap_{ik}\}$. Set the index of the partitions, n=0.

Step 5 Test each of the elements, ap_{in} (n= [1,..,k]), of $AP(S_i)$ for a valid schedule. If there is a valid schedule then stop. Else, store the results of testing each of the partitions in the lists of feasible, F, and infeasible, I, schedules. Once the partition ap_{ik} has been tested and no valid schedule found, the $AP(S_i)$ has been completely examined for the current level of the semilattice.

Step 6 Test the cardinality of the next set system, S_{i+1}, in the list. If $S_{i+1}^{\#}$=CL, then this is an irredundant set system for the same level as that now being tested - go to step 3. If $S_{i+1}^{\#}$ is not equal to CL then all of the irredundants of this level have been tested.

Step 7 Test to see if this is the lowest level of the

semilattice of set systems. If $\#S_i = \#S_1$ then this is the lowest level and there are no redundant set systems. Otherwise all of the redundant set systems and the associated AP's must be formed - go to step 13.

Step 8 Test to see if all of the irredundant set systems have been examined. Is i=q? If "yes", then all of the irredundant set systems have been examined and all remaining set systems are redundant - go to Step 16. If i is not equal to q, then there are still irredundant set systems to be tested for valid schedules.

Step 9 Test to see if the next irredundant set system S_{i+1} has cardinality equal to the number of blocks defined for the next level of the semilattice. If $\#S_{i+1} = CL+1$, then start forming the next level of the semilattice - go to Step 3. If the next irredundant is not in the next level of the semilattice then the semilattice must be "filled in" a level at a time with redundant set systems until the level that contains the next irredundant is reached.

Step 10 Start the formation of the next level. Let CL=CL+1.

Step 11 Form all of the AP's for the redundant set systems by transforming each element of the $AP(S_1)$, l=[1,..i], into the CL block refinements of the partitions. Test each of the partitions for duplication from one set of AP to the other. Retain only one of each distinct partition of the job set. In addition, form only those partitions that may have a valid schedule, i.e., do not form refinements by partitioning any blocks with a feasible schedule.

Step 12 Test each of the ap's for valid schedules. If no valid

schedule exists store any feasible schedules of LCC in the list F and any infeasible schedules in the list I. After all of the partition at this level have been tested and no valid schedule is found, go to Step 9. If a valid schedule is found - stop.

Step 13 This is not the minimal level of the semilattice therefore there exists redundant set systems at this level. The AP's of the redundant set systems with CL blocks are defined by the CL-1 block AP's previously formed. Set the index r=0.

Step 14 Let r=r+1. Form the set of CL block partitions for $AP(S_r)$ from the CL-1 block partitions of $AP(S_r)$. After each partition is formed test to see if it is a duplication of a previously formed partition. If not test for a valid schedule. If there is a valid schedule - stop. If there is no valid schedule augment the lists F and L.

Step 15 Test to see if all of the redundant set systems for this level have been tested. If r=i, then all have been tested - go to Step 8. If r is not equal to i, then go to Step 14.

Step 16 Test to see if all of the possible set system have been formed. If CL=LMAX, then this is the identity element of the semilattice of set systems of LCMC's and further refinements of the partitions is a function of the collection of AP's only. Go to Step 19. If CL is not equal to LMAX, then although all of the irredundant set systems have been formed, there remains elements of the semilattice of set systems, all redundant, yet to be generated.

Step 17 Let CL=CL+1.

Step 18 Form all of the AP's for the CL level of the semilattice

of set systems. Test each of the elements as they are formed for duplication with previously formed partitions, for refinements which are known not to represent a valid schedule, and a valid schedule. If there is no valid schedule augment the lists F and I and go to Step 16. If there is a valid schedule - stop.

Step 19 All of the LCMC's of the job set are contained in the set system last formed. Form the semilattice of AP's a level at a time. Test each of the partitions as they are formed for duplication with previously formed partitions, for refinements which are not feasible, and for a valid schedule. Continue to form the semilattice until a valid schedule is found, once a valid schedule is found - stop.

The algorithm defined above is represented in Figure 4-3.

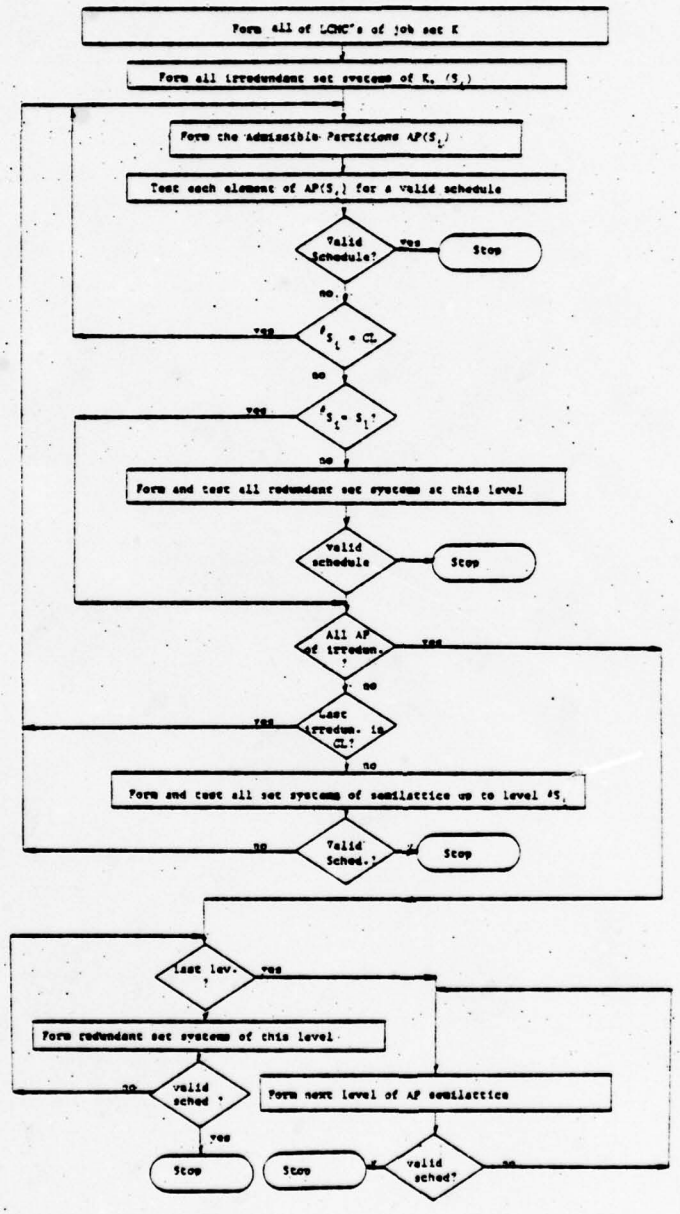


Figure 4-3 Minimal Processor Scheduling Algorithm (Phase II)

4.7 Upper and Lower Bounds for Minimal Processor Schedules

The algorithm defined in the previous section provides for the construction of a minimal processor schedule for a given job set. But, since the algorithm is enumerative in nature, the actual determination of a minimal processor schedule may in fact require more time and computer resources than the system designer desires to expend. As promised in the introduction to this chapter, the algorithm above can be easily modified to include converging upper and lower bounds on the number of processors required, thus enabling the designer to halt the search within a known worst-case difference between the upper bound and a known valid schedule.

We have previously shown that the minimal covering of LCMC's determines the initial lower bound on the number of procesors required. This bound of course increases by one after each of the levels of the semilattice of set systems of LCMC's (or semilattice of AP's) is examined and no valid schedule is determined. Thus after completing an examination of all of the AP's at a given level of the semilattice, say m, it is clear that there can be no fewer than m+1 processors in any valid schedule of the job set.

The upper bound, on the other hand has not been discussed to this point. It too is calculable at every level of the semilattice.

Once the LCMC's of the job set have been determined and before any of the individual LCMC's have been examined to determine if a valid schedule exists, the only known valid schedules for the jobs in the set are determined by the characterization based on the greatest common divisor of the job periods. We know that each of the pairs of jobs within any LCMC have a valid schedule on a single processor; it is

therefore possible to determine an initial upper bound on the number of processors by determining the minimal covering of the job set by valid schedule subsets of the job set K, where valid schedule subset, or valid subset for shorter notation, is defined below.

A valid schedule subset of a job set K, is a maximal subset of an LCMC for which it is known that a valid schedule exists. As an example, consider the LCMC L= {1,2,3,4} for which the only three-job subset with a valid schedule on a single processor is {2,3,4}. The collection of all of the valid schedule subsets of L are {(2,3,4);(1,2);(1,3);(1,4)}.

An integral part of the algorithm defined in the previous section, is the formation of the list of feasible schedules, F. F is in fact the collection of all of the valid schedule subsets of the job set which have been determined up to that time. When the list F is augmented by valid subsets of the LCMC's of one and two job LCC's, the upper bound on the number of processors required for the job set K is the minimal covering of valid schedule subsets.

The determination of the minimal covering of valid subsets is an SCP as described previously.

We will modify the minimal processor scheduling algorithm by adding the determination of a minimal set covering of valid schedule subsets at the completion of each of the levels of the semilattice of LCMC set systems. In addition to defining converging upper and lower bounds on the number of processors required, this procedure will provide a valid schedule with q processors. If the level just completed, m, is equal to q-1, then the schedule found is in fact a minimal processor schedule. Otherwise it defines an upper bound which is within q-m-1

processors of an optimal solution. Thus, this modification to the algorithm may permit the determination of a minimal processor schedule prior to the start of the traversal of the level which contains that AP.

4.8 Summary

In this chapter, we have defined an algorithm to construct a minimal processor schedule for a set of periodic jobs. The algorithm defined is based on the ordering of the possible combinations of the jobs in the set and the optimality of the algorithm defined in Chapter 3. The possible combinations of jobs in the set are ordered such that only collections of Load Consistent Compatibles (LCC's) are considered, and the LCC covers are examined only in order of nondecreasing number of processors required for a valid schedule. The optimality of the uniprocessor scheduling algorithm, Algorithm 3.1, insures that a valid schedule will be constructed for any subset of jobs that has a valid single processor schedule. In addition, there exists the capability to determine a valid schedule which does not contain the minimal number of processors, but is within a known increment of the worst-case bounds of the minimal processor schedule.

CHAPTER 5

SCHEDULING NON-INDEPENDENT PERIODIC TASKS

5.1 Introduction

Throughout the previous chapters we have addressed only the problem of scheduling task systems in which the precedence relationship is empty, i.e. independent tasks. In most task systems including avionic systems, the set of jobs to be scheduled are not mutually independent. In this chapter we will briefly discuss the terminology associated with dependent tasks and its relationship to sets of periodic tasks, and present the relationship between the algorithms developed in the previous chapters for scheduling independent sets of tasks to the problem of scheduling dependent job sets. A brief example of the determination of a valid minimal processor schedule for a non-independent set of jobs will then be presented.

5.2 Task Systems of Non-Independent Jobs

In our original formulation of the problem of scheduling periodic job sets, (Section 2.2), we defined a task system which contained a job set and an associated partial order. Throughout the development that followed, we assumed that the partial order was empty; that is, the jobs were independent. In this section, we will briefly discuss the task system with a non-empty partial order and define ,ecitic terms which relate to such non-independent job sets. We will use the basic terminology of Coffman and Denning (CD).

Let $J=\{j_1, j_2, \ldots, j_n\}$ be a set of jobs with a partial order which

defines the precedence relation on K. The operational precedence specified by the partial order is interpreted as follows:

$j_i \prec j_k$ means that j_i must be completed before j_k is begun.

The job set and its associated precedence relation can be represented graphically by means of a precedence graph. A precedence graph is an acyclic directed graph in which the vertices represent the jobs of K and the the directed edge (j_i, j_k) is in the graph if and only if $j_i \prec j_k$, and there exists no j_p such that $j_i \prec j_p \prec j_k$. One example of a precedence graph of a job set $K=\{j_1, j_2, j_3, j_4, j_5, j_6\}$ is illustrated below.

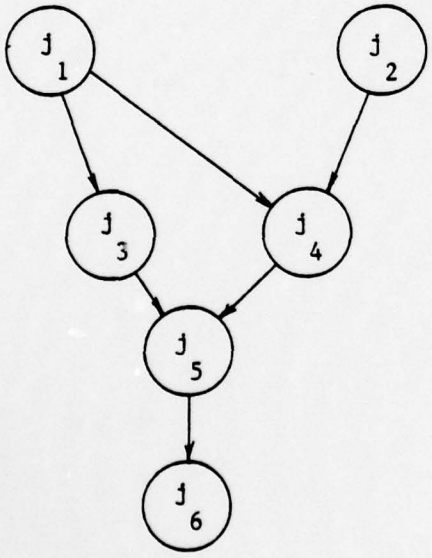


Figure 5-1 A Precedence Graph of Job Set K

Consider a directed path $(j_i, j_k)(j_k, j_p)\ldots(j_q, j_r)$ passing

through the vertices $j_i, j_k, j_p, \ldots, j_q, j_r$. The length of the path is equal to the number of vertices in the path. We say that j_i is a successor of j_k, and j_k is a predecessor of j_i if there exists a directed path from j_k to j_i. If in addition, there is no other job j_p within the directed path of j_k to j_i then j_i is called the immediate successor of j_k and j_k is the immediate predecessor of j_i.

This general formulation of job set precedence relations will be used to define the relationship between the initial active interval of one periodic job and any other periodic job in the set. That is, a job j_i is a predecessor of the periodic job j_k when the initial active interval of j_i must occur prior to the initial active interval of j_k.

In the next section we will discuss the applicability of the previously defined algorithm for multiprocessor scheduling to the case of non-independent job sets.

5.3 Minimal Processor Schedules of Non-Independent Job Sets

In this section we will examine each of the elements of the previously defined algorithms for their applicability to the problem of defining minimal processor schedules for non-independent job sets. First the determination of the LCMC's of a job set is discussed, then the construction of a uniprocessor schedule for a given LCC, and finally the development of a multiprocessor schedule.

Our formulation of the maximal compatibles and the LCMC's of a

job set is dependent only on the execution characteristic of the jobs and not upon any precedence relations among the jobs. It is equally true that any pair of jobs needs to be compatible to be scheduled on the same processor whether they are independent or not. Likewise, no schedule can possibly exist for a given subset of mutually compatible jobs if the load factor exceeds unity.

The formation of schedule equivalence classes for a given LCC of a job set does not take into account any precedence relations among the jobs. The collection of all of the schedule vectors within a given equivalence class of an LCC contains every possible combination of valid schedules for that LCC without regard to precedence constraints. Those schedules within the equivalence class of schedules for a given LCC which satisfy the precedence relations of the job set are a subset, perhaps empty, of the collection of schedules for that LCC if the jobs were mutually independent. Clearly there exists a valid schedule for an arbitrary LCC only if there exists a valid schedule for the LCC if the partial order is empty.

Therefore, the algorithm for construction of a valid uniprocessor schedule of a given LCC, as defined in Chapter 3, must include an additional step in which any valid schedules found must be examined to determine the subset which, if any, also satisfies the precedence constraints of the job set. Satisfying the precedence constraints of a given LCC entails restricting the schedule vectors to those which satisfy the precedence constraints. This is easily done by choosing only those in which the range of the individual schedule elements is appropriatly restricted to represent the precedence relation

of the corresponding job pair. For example, suppose that jobs j_1 and j_2 have periods $T_1=5$ and $T_2=10$ respectively. If j_1 and j_2 are independent, the schedule element s_{12} is defined by the inequality $0 \leq s_{12} \leq [(T_1+T_2)/CI(1,2)]-1$, or $0 \leq s_{12} \leq 2$. On the other hand, if the partial order of the task system required j_1 to precede j_2, then the schedule element s_{12} would have a single value, $s_{12}=0$.

In all other respects the algorithm for constructing a valid uniprocessor schedule is the same and is still optimal in the sense that if there exists a valid uniprocessor schedule it will find one.

As with the construction of a valid schedule on a single processor, the construction of a valid multiprocessor schedule for a set of non-independent jobs depends on the existence of a valid multiprocessor schedule for the same set without consideration of the precedence constraints. But, multiprocessor scheduling includes an added dimension. The precedence constraints of the job set must be satisfied both within the schedules of each of the individual processors and when considered across the schedules of all of the processors. That is, for any jobs j_i and j_k such that $j_i \prec j_k$, the inequality $t_{0i}+E_i \leq t_{0k}$ must be satisfied.

In addition, based on the definition of the tasks given in Section 2.2 (i.e. each task accepts input from a predecessor only when it starts and it provides output only once its execution is completed), a valid schedule is possible for the job set only if for a given path

$(j_i, j_k)(j_k, j_p) \cdots (j_q, j_r)$ the sum of the execution times of every predecessor of any job, say j_p, in the path must not exceed the period of that job minus its execution time. If this is not the case, the job, in this case j_p, will not be able to complete its first execution with all of the required inputs within its required period. Recall again, as an example, the job set used as an example in Section 4.3.

Job	Period	Execution Time
1	1	0.5
2	1	0.4
3	2	1.6
4	2	1.4
5	3	0.5
6	4	0.5
7	6	0.8
8	8	1.5
9	8	0.6
10	9	0.5

Assume that the precedence relationship among the jobs is represented by the graph shown in Figure 5-2.

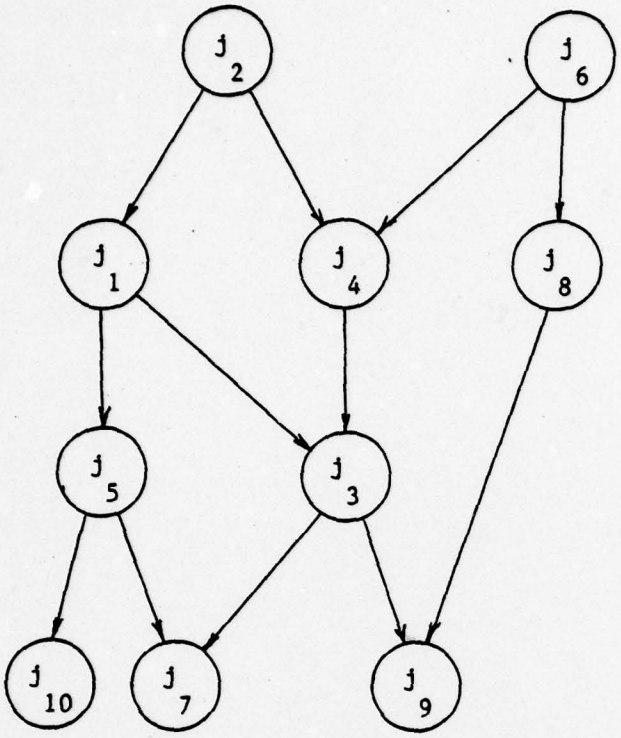


Figure 5-2 Precedence Graph of Example Job Set

There is no valid schedule for the job set with this precedence relationship among the jobs. It can be seen, for instance, that job j_3 with a period of two is a successor of j_4 and j_1 and they are in turn sucessors of j_2. Therefore the start time of j_3 must occur no sooner than $E_2 + E_4$ and $E_2 + E_1$ which are 1.8 and 0.9 respectively for j_3 to satisfy the precedence constraints of the job set. But, if j_3 starts any time later than 0.4, the periodicity requirements of the problem are not satisfied.

The necessity of satisfying the precedence constraints, again, requires adding another step to the algorithms previously defined. Once Algorithm 4.1 or Algorithm 4.2 determines a schedule in which each and every uniprocessor schedule satisfies the precedence constraints of the job set, that schedule must be examined to determine whether the precedence constraints are satisfied across <u>all</u> of the processor schedules. If not, then the search is continued as before. This requirement to satisfy the precedence constraints also impacts that part of the algorithm in which the upper bound is determined, but only with respect to those collections of jobs which are defined to be valid subsets. In this case a valid subset must also satisfy the precedence constraints of the job set, hence the upper bound should converge, in general, more slowly. In any case, the algorithm previously defined is easily adapted to constructing minimal processor schedules for sets of jobs which do not have an empty partial order.

We will now illustrate by example the construction of a minimal

processor schedule for a job set which does not have a null partial order. Consider the job set shown in the previous example and assume a precedence graph as shown in Figure 5-3.

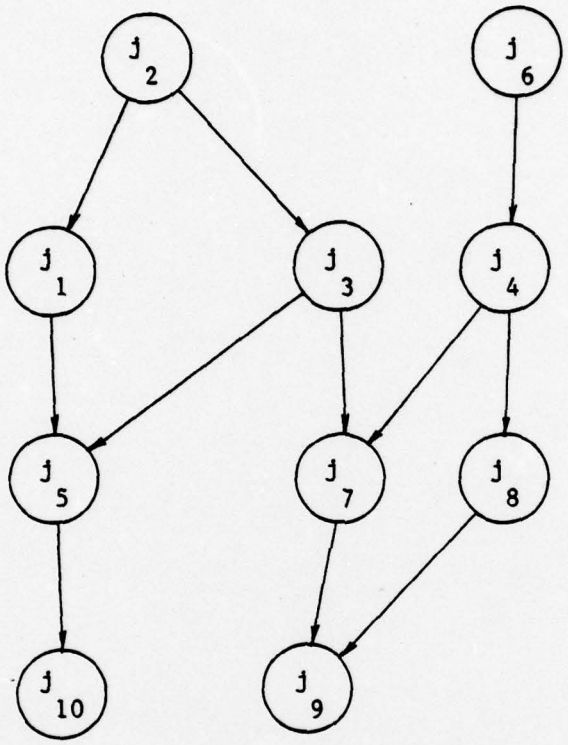


Figure 5-3 Precedence Graph of Example Job Set

In our analysis of the job set with no precedence constraints, (Section 4.3), we constructed a minimal processor schedule

$P_1 = \{j_3\}$

$P_2 = \{j_6, j_8 (0.5), j_9 (2.0)\}$

$P_3 = \{j_4\}$

$P_4 = \{j_5, j_7 (0.5), j_{10} (1.3)\}$

$$P = \{j_5, j_1 (0.5), j_2\}$$

This schedule does not satisfy the precedence constraints of the job set. But, a minimal processor schedule can be found for the job set as represented by the precedence graph of Figure 5-3, as we will illustrate. Without examining in detail the constraints which are not satisfied when the schedules of all of the processors are considered, (e.g. j_3 cannot start at zero and still be a successor of j_2), the constraint of j_2 preceeding j_1 is not satified on P_5.

First let us consider the restrictions that apply to the individual schedule elements of the job set. Table 5-1 lists all the schedule elements of the job set for mutual independence of the jobs and as restricted by the graph above.

Table 5-1 Schedule Elements of Example Job Set

Schedule Elements for Independent Job Set	Schedule Elements for Non-Independent Job Set
$0 \leq s_{12} \leq 1$	$s_{12} = 0$
$0 \leq s_{15} \leq 3$	$0 \leq s_{15} \leq 3$
$0 \leq s_{16} \leq 4$	$0 \leq s_{16} \leq 4$
$0 \leq s_{110} \leq 9$	$1 \leq s_{110} \leq 9$
$0 \leq s_{25} \leq 3$	$1 \leq s_{25} \leq 3$
$0 \leq s_{26} \leq 4$	$0 \leq s_{26} \leq 4$
$0 \leq s_{29} \leq 8$	$1 \leq s_{29} \leq 8$
$0 \leq s_{210} \leq 9$	$1 \leq s_{210} \leq 9$
$0 \leq s_{46} \leq 2$	$s_{46} = 0$
$0 \leq s_{49} \leq 4$	$1 \leq s_{49} \leq 4$
$0 \leq s_{56} \leq 6$	$0 \leq s_{56} \leq 6$
$0 \leq s_{57} \leq 2$	$0 \leq s_{57} \leq 2$
$0 \leq s_{510} \leq 3$	$1 \leq s_{510} \leq 3$
$0 \leq s_{67} \leq 4$	$2 \leq s_{67} \leq 4$
$0 \leq s_{69} \leq 2$	$1 \leq s_{69} \leq 2$
$0 \leq s_{610} \leq 12$	$0 \leq s_{610} \leq 12$
$0 \leq s_{79} \leq 6$	$3 \leq s_{79} \leq 6$
$0 \leq s_{710} \leq 4$	$0 \leq s_{710} \leq 4$
$0 \leq s_{89} \leq 1$	$s_{89} = 1$

The elements of the table above were determined by the periods of the jobs and any precedence relations. For all $T_i \leq T_k$, we know

that $0 \leq s_{ik} \leq [(T_i + T_k)/CI(i,k)]-1$ for independent jobs j_i and j_k which are compatible. Also, $T_i/CI(i,k) \leq s_{ik} \leq [(T_i + T_k)/CI(i,k)]-1$ if and only if $t_{0i} < t_{0k}$, $(j_i \not< j_k)$. Likewise, $0 \leq s_{ik} \leq [T_i/CI(i,k)]-1$ if and only if $j_k \not< j_i$.

For the schedule elements defined above, the schedule P_5 cannot represent the precedence relation defined in Figure 5-3, but since there are only two jobs and they are compatible there exists a valid schedule $\{j_2, j_1(0.4)\}$ that is valid with respect to the precedence constraints. All of the other processors have valid schedules. Now we must examine the validity of the schedule across the set of all processors.

This again is not a valid multiprocessor schedule for the job set. The start times of the jobs j_3 and j_4, for example, do not satisfy the precedence constraints. Fortunately though, a valid schedule is rather easily constructed by shifting the job start times of the individual processor schedules until all of the precedence constraints are satisfied. A valid minimal processor schedule for the job set with the precedence constraints illustrated in Figure 5-3 is:

$P_1 = \{j_3(0.4)\}$

$P_2 = \{j_6, j_8(2.0), j_9(4.5)\}$

$P_3 = \{j_4(0.5)\}$

$P_4 = \{j_5(2.0), j_7(2.5), j_{10}(3.3)\}$

$$P = \{j_5, j_2 (0.4)\}$$
$$\quad\; 1$$

Hence, there exists a valid schedule for this job set which requires five processors - the same number as the minimal processor schedule of the independent job set - but this will not always be the case. Yet, there will never be fewer processors required by a job set with precedence relations added.

Thus we have illustrated the technique of determining for a set of jobs with precedence relations a valid schedule which requires the minimal number of processors using the algorithms developed in the preceeding chapters of this work.

5.4 Summary

In this chapter we have briefly discussed the application of the algorithms previously developed to scheduling non-independent sets of jobs. The consideration of jobs with precedence relations relative to other jobs in the set requires the additional step of determining for each of the valid schedules of the jobs without regard to the precedence constraints whether that schedule satisfies the precedence constraints of the job set, - first with respect to the schedule of each of the individual processors, and secondly with respect to all of the processor schedules.

CHAPTER 6

CONCLUSIONS AND RECOMMENDATIONS

6.1 Introduction

In this chapter we will summarize the development and conclusions of this investigation and briefly discuss possible extensions and recommendations for further work.

6.2 Summary and Conclusions

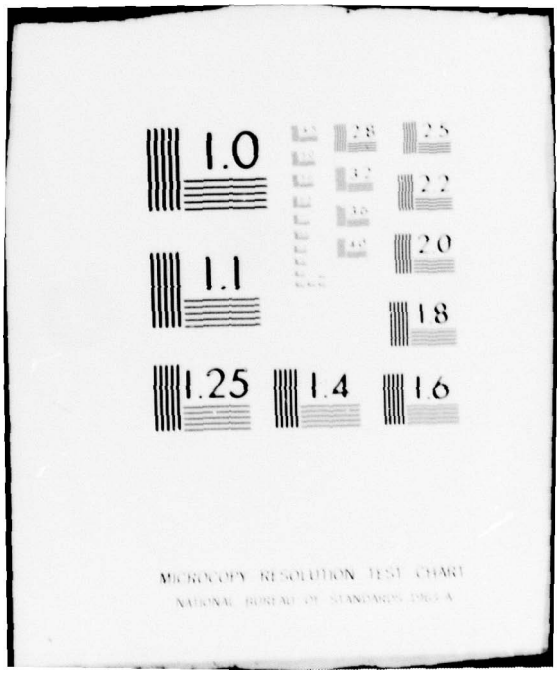
The objective of this investigation has been the development of an algorithm to determine for a given set of periodic jobs with arbitrary integer request periods a valid nonpreemptive schedule that requires the minimal number of processors.

Chapter 2 began with the formal definition of the problem of scheduling a set of periodic tasks; then demonstrated that there exists a simple technique for examining the feasibility of the existence of a valid schedule on a single processor for a subset of jobs. A compatibility realation was defined that determined for each pair of jobs if they excluded each other from existing in a schedule on the same processor. The compatibility relation was based on the relationship of the greatest common divisor (gcd) of the periods of the two jobs to the execution time sum for the two jobs. Only when the sum of the execution times did not exceed the greatest common divisor of the periods were the two jobs compatible and could be scheduled on the same processor. We further restricted the sets of jobs which could be scheduled on the same processor to subsets that were load consistent; that is, those subsets which have a total execution time that does not exceeded unity.

AD-A080 521 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
OPTIMAL MULTIPROCESSOR SCHEDULING OF PERIODIC TASKS IN A REAL-T--ETC(U)
DEC 79 W D SEWARD
UNCLASSIFIED AFIT/DS/EE/79-2 NL

4 OF 4
AD A080521

END
DATE FILMED
3-80
DDC


1.0
1.1
1.25
1.4
1.6
2.8
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

Necessary and sufficient conditions were defined for a valid schedule to exist for any pair of compatible jobs. The results were extended to define for a Load Consistent Compatible (LCC) of arbitrary size the necessary and sufficient conditions for a valid schedule. It was then demonstrated that a lower bound on the number of processors required for a valid schedule of a given job set is determined by the number of elements in a minimal covering of Load Consistent Maximal Compatibles. It was then shown that the construction of a valid schedule for an LCC involved the solution of a mixed integer linear programming problem formulation of the uniprocessor scheduling problem. Next, an equivalence relation on the set of all possible valid schedules was defined, and it was shown to be a relatively simple task to generate for any given consistent schedule its equivalence class. An optimal algorithm for uniprocessor scheduling of periodic jobs using the concepts of equivalence classes of schedules, consistent schedules and relaxation to improve the efficiency of the traversal of the enumeration tree was then formulated. Finally, we briefly discussed the extension of the optimal algorithm to determine optimal schedules of independent jobs based on a linear performance metric.

The algorithm to construct a minimal processor schedule for a set of periodic jobs was developed (Chapter 4). The algorithm was based on the ordering of the possible combinations of the jobs in the set and the optimality of the algorithm developed in Chapter 3. Included in the algorithm is the capability of determining a valid schedule which does not contain the minimal number of processors, but is within a known increment of a minimal processor schedule.

Finally, we briefly discussed, (Chapter 5), the structure of

arbitrary task systems of dependent jobs and their relationship with task systems of periodic jobs. The appliability of the previously developed algorithms to the problem of scheduling non-independent jobs was addressed and a short example of constructing a minimal processor schedule for a set of jobs with a non-empty partial order was presented.

In summary, the major contributions of this investigation are as follows:

- A definition of a compatibility relation which determines for a given set of periodic jobs all subsets that may have a valid schedule.

- An algorithm which is optimal in the sense that it will find a valid uniprocessor schedule for a given set of periodic jobs, if one exists.

- An algorithm for constructing a minimal processor schedule for a given set of periodic jobs.

6.3 Recommendations and Possible Extensions

The recommendations for further development and possible extensions of the results can be divided into two areas: the first area is related primarily to the implementation of the algorithms and the analysis of the solution efficiency in terms of computer time and storage; the other area of work focuses on extensions to the basic formulation of the problem and evaluation of those aspects of the general multiprocessor scheduling problem for periodic tasks.

6.3.1 Implementation and Analysis of Algorithms

The defined algorithms have, to this point, been only partially implemented. The algorithms to form the maximal compatibles and to find the irredundant covers have been completed. The initial segments of the enumeration tree traversal for the uniprocessor scheduling algorithm

have also been completed. Hence, the complete algorithm for determining a minimal processor schedule for a given set of tasks has not been analyzed to determine the efficiency of implementation in terms of computer time and storage requirements. Such an analysis should be undertaken.

The difficulties inherent in analyzing the computational efficiency of algorithms has been noted by Garfinkel and Nemhauser (GN). There is a real problem with dependence on "sample" problems which may be tuned to the advantage of the algorithms in question or may not even represent the class of problems in question. If the algorithm is heuristic, there is little benifit to exercising that algorithm with problem sets unless some measure of the results relative to the optimal solution can be determined. On the other hand, if the algorithm is optimal, computational efficiency should be evaluated using data that describes the problem in question, for example, actual data for an avionic system with a given mission application.

As discussed in the preliminary development of this work, no attempt has been made to ensure that the elements of the total algorithm or any of its phases represent algorithms which are efficient for the general case. For the problem defined in this work, the efficiency of the algorithms which comprise the total is dependent on the characteristics of the individual jobs in the set and their relationship to the other jobs; and a given set of jobs may readily be partitioned into load consistent maximal compatibles (e.g., when each job is compatible with every other job of the set) and yet it is very difficult to determine the final optimal schedule.

In addition, certain elements of the complete multiprocessor

have also been completed. Hence, the complete algorithm for determining a minimal processor schedule for a given set of tasks has not been analyzed to determine the efficiency of implementation in terms of computer time and storage requirements. Such an analysis should be undertaken.

The difficulties inherent in analyzing the computational efficiency of algorithms has been noted by Garfinkel and Nemhauser (GN). There is a real problem with dependence on "sample" problems which may be tuned to the advantage of the algorithms in question or may not even represent the class of problems in question. If the algorithm is heuristic, there is little benifit to exercising that algorithm with problem sets unless some measure of the results relative to the optimal solution can be determined. On the other hand, if the algorithm is optimal, computational efficiency should be evaluated using data that describes the problem in question, for example, actual data for an avionic system with a given mission application.

As discussed in the preliminary development of this work, no attempt has been made to ensure that the elements of the total algorithm or any of its phases represent algorithms which are efficient for the general case. For the problem defined in this work, the efficiency of the algorithms which comprise the total is dependent on the characteristics of the individual jobs in the set and their relationship to the other jobs; and a given set of jobs may readily be partitioned into load consistent maximal compatibles (e.g., when each job is compatible with every other job of the set) and yet it is very difficult to determine the final optimal schedule.

In addition, certain elements of the complete multiprocessor

scheduling problem which are algorithms in their own right, (e.g., the algorithm to determine the maximal compatibles of the job set and the Set Covering and Set Partitioning Problems) (GN), have no known polynomial time implementation. That is, they may be NP-Complete problems themselves.

Therefore, one approach that could be used to analyze the efficiency of the multiprocessor scheduling algorithm would involve implementing the algorithm and applying it to a specific problem. The results of the solution to the problem would then be used to "tune" the algorithm to that application by modifying the separate components of the algorithms and the technique of solution until the overall efficiency of the multiprocessor scheduling algorithm is satisfactory. Such "tuning" might include modifying the search techniques used in the traversal of the semilattice of LCMC set systems and the determination of the maximal compatibles, or the implementation of parts of the algorithm, such as the uniprocessor scheduling algorithm, as a parallel process.

It may also be desirable to investigate the performance of heuristic algorithms for the multiprocessor scheduling problem. Given the ability to determine a minimal processor schedule, the performance of heuristic algorithms could be bounded by the optimal solution.

6.3.2 Generalization of the Problem Characterization

In addition to the extensions which are representative of algorithm implementation questions, there are the other extensions which are related to the generalization of the problem statement. Among the possible generalization are the inclusion of heterogenous processors,

processor system architectures that are dependent on other than minimal number of processors, and the limitation to jobs with integer execution times. We will consider each of these possible extensions in order.

The problem considered in this investigation defined all of the processors to be identical in processing capability. In many environments this may not be the case. For instance, a system may require special purpose processors to perform high speed signal processing. While such a machine would not in general accommodate any other tasks of the job set, the synchronization of this processor with other tasks might impact the overall schedule of the job set. As another example, many processes require that some of the execution be performed at a higher rate than that of other processes. Additional work should be done in the area of scheduling a set of jobs on a heterogenous set of processors where there exists known subsets of each of the different processors available for use in the system.

We have addressed only the problem of finding the minimal processor schedule. In most actual systems there are other aspects that are important in defining the system that is best for the application. The bus structure of a multiprocessor system effects both the interprocessor communication and the ability of the system to be reconfigured so that the system will continue to function in a degraded mode. The analysis of the scheduling should be extended to include the loading on the system data bus as well as the minimal processors schedules. Much of this can be accomodated within the algorithms developed by making use of the linear constraint function of the linear programming algorithm.

Finally, although we have "generalized" our problem by admiting

all non-negative real numbers as possible execution times for the jobs, with any computer application the execution times must each require an integer number of basic cycles of the processor. With integer periods and execution times for each of the jobs, it is possible that there could be a reduction in the computation required to determine the existence of a valid uniprocessor schedule for a given LCC of jobs. Clearly, when there are two jobs there exists a valid schedule if and only if the is a compact assignment of the jobs. Likewise by "shifting" the active intervals of any set of job which have a valid single processor schedule, it can be shown that if there is a valid schedule for the jobs there is a schedule in which there is an adjacent assignment for each of the jobs. If these properties can be extended to show that there is a valid schedule for any set of jobs if and only if there is at least one adjacent assignment for each of the jobs in the set, then the total number of possible assignments for any set of jobs could be reduced significantly.

6.4 Final Remarks

Although there exists much work left to be done in the area of scheduling periodic tasks, it is hoped that the results of this investigation will inspire others to further examine the development of both the theoretical and applications aspects of the problem.

BIBLIOGRAPHY

(Aro) Aron, J.D., "Real-time Systems in Perspective," Real-Time Systems Design, Yourdan, E., Ed., Paragon Press, Somerville Mass., 1967, pp. 7-28.

(ATT) Auslander, D.M., et al, "Direct Digital Process Control: Practice and Algorithms for Microprocessor Application," Proceedings of the IEEE, Vol 64, No. 6, Jun 1976, pp. 199-208.

(BA) Bobrow, L.S. and M. A. Arbib, Discrete Mathematics Applied Algebra for Computation and Information Science, Saunders, Phil. Pa., 1974.

(Ber) Berztiss, A.T., Data Structures: Theory and Practice, Academic Press, New York, 1975.

(BF) Berfield, R. G. and Felsman, "Integrated Inertial Doppler LORAN Computer Guidance and Control," Computers in the Guidance and Control of Aerospace Vehicles, Leondes, Ed., AGARDograph 158, 1972, pp. 157-174.

(BGJ) Brucker, P., et al., "Scheduling Equal Length Tasks Under Treelike Precedence Constraints to Minimize Maximum Lateness," Mathematics of Oper. Res., Vol 2, No. 3, Aug 1977.

(Bib) Bibbero, R.J., Microprocessors in Instruments and Control, John Wiley and Sons, New York, 1977.

(Bir) Birkhoff, G. and T.C. Bartee, Modern Applied Algebra, McGraw-Hill, New York, 1970.

(Boo) Booth, T.L., Sequential Machines and Automata Theory, John Wiley and Sons, New York, 1967.

(CD) Coffman, E.G. and R.J. Denning, Operating Systems Theory, Prentice-Hall, Englewood Cliffs, N.J., 1973.

(Cha) Chamberlin, L.A., et al., Design of the Core Elements of the Digital Avionics Information System (DAIS), Vol II, AFAL-TR-74-245, 1974.

(Chr) Christofides, N., Graph Theory: An Algorithmic Approach, Academic Press, New York, 1975.

(CMM) Conway, R. W., et al., Theory of Scheduling, Addison-Wesley, Reading Mass., 1967.

(Cof) Coffman, E.G., Ed., Computer and Job/Shop Scheduling Theory, John Wiley and Sons, New York, 1976.

(Dan) Dantzig, G.B., Linear Programming and Extensions, Princeton Univ. Press, Princeton, N.J., 1963.

(Das) Das, S.R., "On a New Approach for Finding All the Modified Cut-Sets in an Incompatibility Graph," IEEE- Transaction on Computers, Vol c-22, No. 2, Feb. 73, pp. 187-193.

(Dha) Dhall, S.K., "Scheduling Periodic -Time-Critical Jobs on Single Processor and Multiprocessor Computing Systems," Ph.D. Dissertation, Univ. of Illinois, Urbana-Champaign, Ill., 1977.

(DL) Dhall,S.K. and C.L. Liu, "On a Real-Time Scheduling Problem," Operations Research, Vol 26, No. 1, Jan-Feb 1978, pp. 127-140.

(Fis) Fisher, J.L., Application-Oriented Algebra, A. Dun-Donnelley, New York, 1977.

(Ful) Fuller, S.H., et al., "Multi-Microprocessors: An Overview and Working Example," Proc. of IEEE, Feb. 1978, pp. 216-228.

(Gas) Gass, S.I., Linear Programming, McGraw-Hill, New York, 1975.

(Gel) Gelb, A., Applied Optimal Estimation, MIT Press, Reading Mass, 1974.

(Geo) Geoffrion, A.M. and R.E. Marsten, "Integer Programming Algorithms: A Framework and State-of-the-Art Survey," in: Perspectives on Optimization, Geoffrion Ed., Addison-Wesley, New York, 1972.

(GGJ) Garey, M.R., et al., "Performance Guarantees for Scheduling Algorithms," Operations Research, Vol 26, No. 1, Jan. 78, pp. 3-21.

(GJ1) Garey, M.R. and D. Johnson, "Complexity Results for Multiprocessor Scheduling Under Resources Constraints," SIAM J. Comp, Vol 4, No. 4, Dec 75, pp. 397-411.

(GJ2) "____.____", "Scheduling Tasks with Nonuniform Deadlines on Two Processors," JACM, Vol 23, NO. 3, Jul 1976, pp. 461-467.

(GJ3) "____."____", "Two-Processor Scheduling with Start-Times and Deadlines," SIAM J. Comput., Vol 6, No. 3, Sept 1977, pp. 416-426.

(GN) Garfinkel, R.S. and G.L. Nemhauser, Integer Programming, John Wiley and Sons, New York, 1972.

(Gon2) Gonzalez, M.J., "Deterministic Processor Scheduling," Computing Surveys, Vol 9, No. 3, 1977.

(Gon1) "____", "Problem Areas in Distributed Processor System Design," Proc. Milwaukee Symposium on Automatic Computation and Control, 1976, pp. 161-166.

(GS) Gonzalez, M.J. and J.W. Soh, "Periodic Job Scheduling in a Distributed Processor System," IEEE Trans. on Aerospace and Electronic Systems, Vol Aes-12, No. 5, Sept. 1976, pp. 530-535.

(Har) Hartmanis, J. and R.E. Stearns, Algebraic Structure Theory of Sequential Machines, Prentice-Hall, 1966.

(Jen) Jensen, E.D., "The Honeywell Experimental Distributed Processor - An Overview," Computer, Jan 1978, pp. 28-38.

(Joh) Johnson, et al., All Semiconductor Distributed Processor/Memory Study, Vol II, AFAL-TR-73-226, 1973.

(Jos) Joseph, E., "Innovations in Heterogeneous and Homogeneous Distributed Function Architectures," Computer, Mar 1974, pp. 17-24.

(Kar) Karp, R.M., "Reducibility Among Combinatorial Problems," in: Complexity of Computer Computation, Miller and Thatcher Ed., Proc. IBM Symposium, Mar 20-22, 1972, Yorktown Heights, N.Y.

(Kil) Kilpatrick, P.S., et al., All Semiconductor Distributed Aerospace Processor/Memory Study, Vol I, Part 1, AFAL-TR-74-245, 1974.

(Kis) Kise, H., et al., "A Solvable Case of the One-Machine Scheduling Problem with Ready and Due Times," Operations Research, Vol 26, No. 1, Jan-Feb 78, pp. 121-126.

(Lab) Labetoulle, J., "Some Theorems on Real Time SCheduling," in: Computer Architecture and Networks, pp. 285- 298, Gelenbe and Mahl, Ed., North Holland Pub Co., 1974.

(Liul) Liu, C.L., Topics in Combinatorial Matematics, Math. Assoc. of America, 1972.

(Liu2) "____", Elements of Discrete Mathematics, McGraw-Hill, New York, 1977.

(LL) Liu, C.L. and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," J. ACM, Vol 20, No. 1, Jan 1973, pp. 46-61.

(Mar) Martin, J., Design of Real-Time Computer Systems, Prentice-Hall, Englewood Cliffs, N.J., 1967.

(MF) Mcmahon, G. and M. Florean, "On Scheduling with Ready Times and Due Dates to Minimize Maximum Lateness," Operations Research, Vol 23, No. 3, May-Jun 1975.

(Mo) Moore, J.M., "An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs," Management Science, Vol 15, No. 1, Sept. 1968.

(Pra) Prather, R.E., Introduction to Switching Theory: A Mathematical Approach, Allyn and Bacon Inc., 1967.

(Rin) Rinnooy Kan, A.H.G., et al., "Minimizing Total Costs in One-Machine Scheduling," Operations Research, Vol 23, No. 5, Sept-Oct 75.

(Saa) Saaty, T.L., Optimization in Integers and Related Extremal Problems, McGraw-Hill, New York, 1970.

(Sav) Savage, J.E., The Complexity of Computing, John Wiley and Sons, New York, 1976.

(Ser) Serlin, O., "Scheduling of Time-Critical Processes," Proc. SJCC (1972), Vol 40, pp. 925-932.

(SF) Searle, B.C. and D.E. Freberg, "Tutorial: Microprocessor Applications in Multiple Processor Systems," Computer, Oct 1975, pp. 22-30.

(SH) Shani, S. and Horowitz, E., "Combinatorial Problems: Reducibility and Approximation," Operations Research, Vol 26, No. 5, Sept-Oct 1978.

(Sha) Shani, S. K., "Algorithms for Scheduling Independent Tasks," JACM, Vol 23, No. 1, Jan 76, pp. 116-127.

(Sid) Sidney, J.B., "Optimal Single-Machine SCheduling with Earliness and Tardimess Penalties," Operations Research, Vol 25, No. 1, Jan-Feb 1977.

(Sie) Sierpenski, W., Elementary Theory of Numbers, Panstwowe Wydawnictwo Naukowe, Warszawa Poland, 1964.

(Soh) Soh, J. W., "Scheduling Strategies for Periodic Jobs in a Multiprocessor Environment," Ph.D Dissertation, Northwestren, Univ., Evanston, Ill., 1974.

(Ste) Stewart, B.M., Theory of Numbers, Macmillan Co., New York, 1964.

(Ull) Ullman, J.D., "NP-Complete Scheduling Problems," JCSS, Vol 10, No. 3, June 1975.

(US) Unger, F.G. and Sindlinger, "Systems Tasks for Advanced Aircraft Navigation Systems," Computers in the Guidance and Control of Aerospace Vehicles, Leondes, Ed., AGARDograph 158, 1972, pp. 117-130.

(Yan) Yang, C.C., "On a New Approach for Finding All Modified Cut-Sets in an Incompatibility Graph," IEEE-Transaction on Computers, Vol c-24, No. 10, Oct 75, pp. 977-979.

APPENDIX A

AN INDEX OF DEFINITIONS AND TERMINOLOGY

This appendix contains an index to the definitions and terminology used thoughout this investigation. The page associated with each term refers to the page where the term is defined or first used in the text.

	page
accessible regions	97
active interval	14
active period	14
admissible partition (AP)	242
candidate problem (CP)	133
chain-connected sets	243
closed collection of schedule elements	187
closed set of schedule elements	187
common divisor	47
compact in the initial interval	38
compact schedule	38
compatible jobs	96
compatibility class	70
compatibility relation	96
conflict	11
connected sets	243
consistent enumeration tree	189
consistent tree	189
cover of a set	206
cover table	212
critical interval (CI)	107
current frame of a task	94
deadline-driven scheduling algorithm	94
deterministic scheduling	10
Diophantine Equation	52
directed path	263
efficient algorithm	11
empty processor schedule	37
Essential Load Consistent Maximal Compatible (ELCMC)	213
essential subset	218
family of schedules	119
feasible schedule	10
fixed priority algorithm	15
frequency decreasing priority	21
Gantt chart	13
greatest common divisor	47

homogeneous processor set 8
idle interval 14
idle period 14
immediate predecessor 264
immediate successor 264
inconsistency (Level-1/Level-2) 170
independent jobs/tasks 8
initiation time list 108
intelligent fixed priority algorithm 95
irredundant covering 211
job 8
job active interval 13
job active period 13
job idle interval 13
job idle period 13
job start time 36
job time sequence 35
length of the critical interval 103
lexicographic ordering 33
Load Consistent Compatibility Class (LCC) 89
Load Consistent Maximal Compatible (LCMC) 89
load factor 32
maximal compatible 70
minimal covering 207
nonpreemptive scheduling 10
N-P Complete 11
NP-Hard 18
optimal algorithm 92
precedence graph 263
predecessor job 264
preemptive scheduling 10
preorder traversal 80
processor job schedule 37
processor time sequence 37
"pseudo" job 109
Rate-Monotonic-First-Fit Scheduling (RMFFS) 16
Rate-Monotonic-Next-Fit Scheduling (RMNFS)..... 16
rate monotonic ordering 16
Rate-Monotonic Priority Algorithm 95
reference job 43
refinement of a partition 242
"reflected" active interval 111
relation matrix 70
relative urgency algorithm 94
restricted region 39
schedule conflict 11
schedule element 126
schedule inconsistency 168
schedule vector 139
schedule vector sequence 165
semilattice 236
separation of candidate problem 197
sequence of reference jobs 142
sequence of schedule elements 142

set-covering 207
set covering problem (SCP) 207
set partitioning 207
set system 234
static scheduling algorithm 15
stochastic scheduling 10
subschedule 57
subsequence of schedule elements 143
successor job/task 264
successor job set 218
task 8
time-critical process 13
time-critical task 13
valid schedule 10
valid schedule subset 260
valid subset 260

APPENDIX B

GROUP THEORETIC NOTATION, DEFINITIONS, AND THEOREMS

The notation, definitions, and theorems presented in this appendix can be found in any text on group therory (BA, Bir, Liu2, Fis). This is not intended to be a complete development of the theory of groups and no proofs of the theorems will be presented here. Only the results necessary to support the work of this investigation will be included. Furthermore, since the groups used in this investigation are commutative, we have used the notation applicable to commutative groups instead of the more general notation usually employed to present general group theoretic results.

The theorems listed here are general results and can be found in all of the references listed above except as noted.

A nonempty set G together with a binary operator + on G is a group, (G,+), if:

1. $(g_1+g_2)+g_3 = g_1+(g_2+g_3)$ for all g_1, g_2, g_3 in G. That is, the operator + is associative.

2. There is an identity 0 in G such that 0+g=g=g+0 for all g in G.

3. Each element g in G has an inverse -g in G such that g+(-g)=0= -g+g.

There are in addition relationships that may exist between the group and certain subsets of the group. These subsets may be groups in their own right and give rise to a natural partition of the group.

A nonempty subset H of a group (G,+) is a subgroup of (G,+) if (H,+) is a group under the same binary operation +.

The following theorem defines the requirements for a given finite subset of a group to form a subgroup.

Theorem B.1 Let (G,+) be a group and H a subset of G. If H is a finite set, then (H,+) is a subgroup of (G,+) if + is a closed operation on H (Liu2).

The groups upon which the proofs of this investigation are based are all finite. Therefore, any subset of these groups are subgroups if they satisfy the requirements of Theorem B.1.

There are further refinements to the collection of groups of interest to us in this investigation.

A group (G,+) is an abelian group if, in addition to the requirements above, the operation + is commutative. That is, $g_1+g_2 = g_2+g_1$ for all g_1 and g_2 in G.

The power of any element g in G are defined for the nonnegative integers n=[0,1,2,3...) by recursion as follows:

$0g=0$, $1g=g$, $2g=g+g,\ldots,(n+1)g=ng+g$ (Bir).

A group (G,+) is called cyclic when it consists of the powers ng of some one of its elements g. In addition, we say that the set $\langle g\rangle = \{ng \text{ for all integer } n\}$ generates a subset of G. The element g is termed a generator of G.

There are certain properties that pertain to cyclic groups and any of their subgroups.

Theorem B.2 A subgroup of a cyclic group is a cyclic group.

Theorem B.3 Every cyclic group is abelian.

Let g be an element of G and H a subset of G. The coset of H with respect to g, which we will denote as g+H, is the set of elements {g+x : x an element of H}. (Note, for groups in general there are defined both left and right cosets, but since we are dealing with abelian groups only, the left and right cosets are equal.)

Theorem B.4 The set of distinct cosets of a subgroup (H,+) of the group (G,+) partitions G.

Theorem B.5 If (H,+) is a finite subgroup of (G,+), then the coset g+H contains the same number of elements as H.

VITA

Walter D. Seward was born on 5 October 1942 in Las Vegas, New Mexico. He graduated from Robertson High School, Las Vegas, New Mexico, in 1960, and then entered New Mexico State University where he was elected to Eta Kappa Nu. In 1967, he received the degree of B.S.E.E., and was commissioned as a Second Lieutenant in the United States Air Force.

From 1967 to 1969, he attended the Air Force Institute of Technology and received an M.S.E.E. degree and was elected to Tau Beta Pi. He was then assigned to the Air Force Flight Test Center, Edwards, AFB, California. At AFFTC, he served as a Project Engineer and Section Chief of tHe hybrid Computer Simulation Laboratory, working with real-time simulations of aircraft, primarily the X-24 lifting body.

In 1973, he entered the Air Force Institute of Technology.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/DS/EE/79-2	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Optimal Multiprocessor Scheduling of Periodic Tasks in a Real-Time Environment		5. TYPE OF REPORT & PERIOD COVERED Ph.D. Dissertation 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Walter D. Seward Capt, USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFAL/AAF Air Force Avionics Laboratory Wright-Patterson AFB, OH 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 2003 03 20
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/ENE Air Force Institute of Technology Wright-Patterson AFB, OH 45433		12. REPORT DATE December 1979 13. NUMBER OF PAGES 306
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

JOSEPH P. HIPPS, Maj, USAF
Director of Public Affairs

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Scheduling Theory; Digital Computer Systems; Multiprocessor Systems; Periodic Tasks; Optimal Algorithms; Computer Scheduling; Nonpreempted Schedules; Minimal Processor Schedules.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The problem of constructing a nonpreemptive schedule that requires the minimal number of processors for a given set of periodic tasks is examined. Each periodic task is characterized by an integer period and an execution time. It is assumed that the period between each initiation and termination of a task must not vary once the first is specified.

A compatibility relation is defined on the set of tasks such that any pair

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Item 20 continued:

of tasks may be scheduled on the same processor only if they are compatible. This compatibility relation is based on the greatest common divisor of the task periods and the sum of the task execution times. An equivalence class of schedules of a given subset of tasks is defined and an optimal algorithm for constructing a single processor schedule is developed. This algorithm is optimal in the sense that it will construct a single processor schedule if one exists.

A lower bound on the number of processors required for the given task set is defined based on the minimal covering of maximal compatibles of the task set. A lattice structure is then developed from all of the irredundant coverings of maximal compatibles. The lattice structure contains every possible combination of tasks which may have a valid schedule. An algorithm is defined which constructs a minimal processor schedule for a given set of tasks. This algorithm is based on the optimal single processor algorithm and the implicit enumeration of the possible schedules. The algorithm determines both upper and lower bounds on the number of processors required. These bounds converge to a common value as the algorithm converges on an optimal schedule.