

DTIC FILE COPY



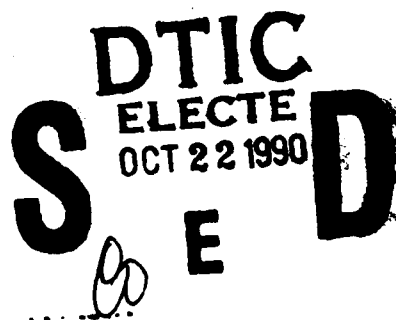
RADC-TR-90-182
Final Technical Report
August 1990

AD-A227 856

DECENTRALIZED REAL-TIME SCHEDULING

Carnegie Mellon University

J. Duane Northcutt, Raymond K. Clark, David P. Maynard, Jeffrey E. Trull



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

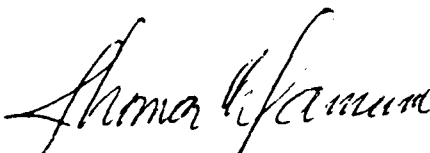
Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

90 10 18 022

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

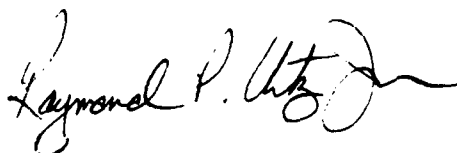
RADC-TR-90-182 has been reviewed and is approved for publication.

APPROVED:



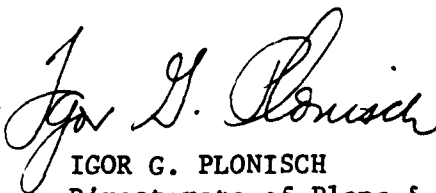
THOMAS F. LAWRENCE
Project Engineer

APPROVED:



RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:



IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE August 1990	3. REPORT TYPE AND DATES COVERED Feb 89 to Feb 90		
4. TITLE AND SUBTITLE DECENTRALIZED REAL-TIME SCHEDULING		5. FUNDING NUMBERS C - F30602-88-D-0026, Task B-9-3505 PE - 62702F PR - 5581 TA - 21 WU - PG		
6. AUTHOR(S) J. Duane Northcutt, Raymond K. Clark, David P. Maynard, Jeffrey E. Trull				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Carnegie Mellon University School of Computer Science Dept. of Electrical and Computer Engineering Pittsburgh PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER N/A		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Air Development Center (COTD) Griffiss AFB NY 13441-5700		10. SPONSORING/MONITORING AGENCY REPORT NUMBER RADC-TR-90-182		
11. SUPPLEMENTARY NOTES RADC Project Engineer: Thomas F. Lawrence/COTD/(315) 330-2158				
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.		12b. DISTRIBUTION CODE		
13. ABSTRACT (Maximum 200 words) > This document describes experiments designed to evaluate the behavior and performance of various scheduling policies for the Alpha real-time distributed operating system. (Northcutt 87): Alpha is an adaptable decentralized operating system being developed as a part of the Archons project's on-going research into real-time distributed systems. Timely completion of an application's computational activities is one of the most important functions of a real-time system. Therefore, the proper scheduling of those activities is of critical importance. The Alpha programming model provides simple mechanisms for an application to specify its timeliness requirements. The scheduler may consider this time constraint information when scheduling activities for execution. To gauge the effectiveness of Alpha as an operating system for real-time applications, an understanding of its scheduling facility is necessary. To this end, a study of the effects of various different scheduling policies was made. To evaluate these policies, a real-time application was created to serve as an experimental workload. The application was				
14. SUBJECT TERMS Real-Time Scheduling Distributed System Decentralized Control		Distributed Operating System Alpha Best-Effort Scheduler		15. NUMBER OF PAGES 246
				16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

UNCLASSIFIED

designed to simplify the modeling of tasks with a wide range of timeliness requirements and to provide a visual indication of scheduler performance.

Alpha provides a clean, well-defined interface between the scheduling subsystem and the kernel proper. A general scheduler framework, independent of any specific policy, has been implemented to simplify the development and substitution of different policy modules. This framework greatly simplifies the experimental comparison of different policies. To date, eight different scheduling policies have been implemented for Alpha. Five of these policies are examined in this document - preemptive round-robin, static priority, dynamic deadline, shortest processing time, and Best-Effort (Locke 86).

The experiments were designed to compare the application-level behavior of various scheduling policies under a variety of loading conditions. Each policy was judged based on criteria such as how well the application timeliness requirements were met and what fraction of the potential value of the application was obtained. The experimental results were then studied to develop empirical observations about the relative behavior of each scheduling policy and to derive an understanding of the scheduling decisions that were made.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



UNCLASSIFIED

Preface

The research discussed in this report investigates decentralized real-time scheduling in the context of the Alpha Operating System. Four goals have been identified for this work: (1) to identify areas where Alpha would benefit from additional decentralized real-time scheduling research; (2) to define decentralized real-time scheduling algorithms for incorporation into Alpha; (3) to develop criteria to determine the efficacy of these algorithms; and (4) to evaluate the algorithms.

A two-pronged approach has been adopted to satisfy these goals. On one front, Alpha Release 1 is studied and evaluated. This effort identifies and evaluates the relevant real-time scheduling facilities in Alpha, revealing open questions that require further research. On the second front, research is performed that will improve Alpha in future releases. In some cases, this work focuses on questions identified by the analysis of Alpha Release 1, while in other cases it extends Alpha's capabilities in directions previously identified—for example, providing the foundation for the real-time transaction facility. In the future, the results of this research will be put into practice in Alpha.

This report is divided into three sections, one evaluating Alpha Release 1 and the others reporting on new scheduling research.

Part A describes a demonstration program that was used to study various scheduling policies, including a Best-Effort Scheduler, in the context of Alpha Release 1. The program was designed specifically to isolate scheduling policy decisions, to provide a graphic display of scheduling algorithm behavior, and to measure scheduler performance. Each scheduling policy was analyzed under a variety of workloads, with Alpha's Best-Effort Scheduler performing well. The results of this investigation are included in Part A.

Part B addresses the issue of scheduling distributed computations. It outlines ongoing work to model decentralized and distributed computations, to describe best-effort scheduling algorithms appropriate for this model, and to study the amount and type of information that is required to schedule the computations effectively.

Part C extends best-effort techniques to schedule dependent activities in a supervisory control real-time system. Dependent activities, which share common resources such as data and devices, may also exchange signals to coordinate their actions. This study provides an analytic framework to formally investigate various scheduling policies. Several useful properties of the resulting scheduling algorithm are proven within this framework. In addition, simulation results are presented that demonstrate the effectiveness of the algorithm when compared to other algorithms. (Part C is actually a draft of a doctoral dissertation. As such, it is expected that it will be revised and augmented in order to produce the final version of the dissertation.)

Note: This Final Technical Report and the Six-Month Technical Report (Interim Techni-

cal Report) together detail the research performed for this contract. While the material presented in Part C of this report updates and expands on Part A of the Interim Technical Report; Part B of the Interim Technical Report, which described the completed analysis of the Communication Subsystem of Alpha Release 1, is not repeated or revised in this document.

The Alpha Operating System: Scheduler Evaluation Experiments

**Jeffrey Trull
J. Duane Northcutt
David P. Maynard
Raymond K. Clark**

*School of Computer Science
Department of Electrical and Computer Engineering
Carnegie Mellon University*

February 6, 1990

Table of Contents

Abstract.....	A-1
1. Introduction.....	A-3
1.1 The Alpha Operating System.....	A-3
1.1.1 Real-Time Scheduling Requirements	A-4
1.1.2 Distribution Requirements	A-5
1.2 Testing Alpha Scheduling Policies	A-6
1.2.1 Test Objectives.....	A-6
1.2.2 Test Application Requirements.....	A-7
2. Scheduler Evaluation System	A-8
2.1 Application Program Structure	A-8
2.1.1 UNIX Implementation	A-11
2.1.2 Alpha Implementation	A-11
2.2 External Environment Simulator	A-12
2.2.1 Simulator Structure	A-12
2.2.2 Operator Interface	A-12
3. The Structure of an Alpha Scheduler	A-15
3.1 Scheduler/Kernel Interface	A-15
3.2 Structure of Generic Scheduler.....	A-17
4. Scheduling Policies.....	A-19
4.1 Round Robin	A-19
4.2 Static Priority	A-19
4.3 Deadline	A-19
4.4 Shortest Processing Time.....	A-20
4.5 Best Effort.....	A-20
5. Experimental Results.....	A-21
5.1 Experimental Design.....	A-21
5.1.1 Load Generation.....	A-21
5.1.2 Evaluation Metrics	A-21
5.2 Behavior Analysis.....	A-22
5.3 Simulation Results	A-26
5.3.1 Thread Importance Sensitivity	A-26
5.3.2 Meeting Application Time Constraints.....	A-29
5.3.3 Maximizing Application Value.....	A-30
6. Conclusions.....	A-34
References	A-35

Abstract

This document describes our experience with a set of experiments designed to evaluate the behavior and performance of various scheduling policies for the Alpha real-time distributed operating system [Northcutt 87]. Alpha is an adaptable decentralized operating system being developed as a part of the Archons project's on-going research into real-time distributed systems. Timely completion of an application's computational activities is one of the most important functions of a real-time system. Therefore, the proper scheduling of those activities is of critical importance.

The Alpha programming model provides simple mechanisms for an application to specify its timeliness requirements. The scheduler may consider this time constraint information when scheduling activities for execution. To gauge the effectiveness of Alpha as an operating system for real-time applications, an understanding of its scheduling facility is necessary. To this end, a study of the effects of various different scheduling policies was made. To evaluate these policies, a real-time application was created to serve as an experimental workload. The application was designed to simplify the modeling of tasks with a wide range of timeliness requirements and to provide a visual indication of scheduler performance.

Alpha provides a clean, well-defined interface between the scheduling subsystem and the kernel proper. A general scheduler framework, independent of any specific policy, has been implemented to simplify the development and substitution of different policy modules. This framework greatly simplifies the experimental comparison of different policies. To date, eight different scheduling policies have been implemented for Alpha. Five of these policies are examined in this document—preemptive round-robin, static priority, dynamic deadline, shortest processing time, and Best-Effort [Locke 86].

The experiments were designed to compare the application-level behavior of various scheduling policies under a variety of loading conditions. Each policy was judged based on criteria such as how well the application timeliness requirements were met and what fraction of the potential value of the application was obtained. The experimental results were then studied to develop empirical observations about the relative behavior of each scheduling policy and to derive an understanding of the scheduling decisions that were made.

1. Introduction

This document describes the experience of the Archons research project in evaluating a set of scheduling policies for use in the Alpha distributed real-time operating system. Alpha is unique among operating systems in the problem area it seeks to address: distributed, supervisory-level, real-time command and control. This problem area dictates some special requirements that the system must fulfill. These requirements are associated chiefly with the need to manage resources in a timely and decentralized fashion. The Alpha programming model permits the convenient expression of the timeliness requirements of an application. The manner in which the active scheduling policy uses this information determines how well the timeliness requirements are satisfied. This set of experiments examines the behavior and performance of five different scheduling policies that have been implemented for Alpha. Before examining these policies or the experiments in detail, however, it is useful to understand the nature and requirements of Alpha.

1.1 The Alpha Operating System

The Alpha operating system is unique because of the requirements imposed by its application domain. As a distributed real-time command and control system, Alpha is intended to support applications where most of the activities in the system have stringent time constraints that are a matter of correctness rather than convenience—the safety of human life and property may be dependent on the correct functioning of the system. In addition, the distributed nature of the system also places unusual demands on Alpha. Even though the system consists of a collection of physically dispersed processing elements, system computation resources must be managed so as to cooperatively perform a single mission, rather than treated (as is currently common) as a network of communicating, but otherwise independent and unrelated individual processing elements.

The Alpha programming model directly supports these needs [Northcutt 88b]. From the viewpoint of the programmer, the physically dispersed system may be logically viewed as a centralized one. The basic abstractions in Alpha include *objects*, *threads*, and *operation invocations*. In Alpha, an object is defined as a logically related collection of data and the code used to manipulate that data. The external interface to an object consists of the set of operations that may be performed on the object, and represents the only means of accessing the object's encapsulated data. The basic unit of activity in the Alpha system is known as a *thread*, which is a representation of a logical point of program execution. Threads are the entities which animate the otherwise passive objects in the system. Threads move between objects by means of operation invocations. Operation invocations are similar in some ways to procedure calls, accepting and returning parameters in much the same way.

In Alpha, there may be any number of threads executing within a single object. Furthermore, because objects may be on any physical processing element, threads may, as a result, move between the processing nodes in the system. When a thread is created, it begins its execution with the invocation of some operation on an object. The thread continues execu-

tion until this initial operation is complete, at which point it is deleted. In the process of executing any operation, a thread may invoke operations on other objects, thereby transferring the thread into a new object (which may be on a different node). If the target of an invocation is on a different node, the thread state (including the current time constraints) is transferred to the destination node.

It is important to recognize the difference between this programming model and that of systems which support the *process-message* model. In the process model, a unit of computational activity is tightly bound to a piece of code and a physical processing element. When an application is organized into objects which may be used by many system activities, it does not make sense to confine points of control to specific code segments which belong exclusively to each thread. The thread abstraction in Alpha represents the essence of an abstract computational activity, and is not burdened by unnecessary artifacts. A thread represents a locus of program control and the current attributes the activity (e.g., its time constraints). Threads move between objects without regard to their physical location, and do not incur scheduling overhead as a result of each inter-object transition.

With respect to the real-time objectives of the system, the single most important resource that the Alpha system manages is the processing cycles available to the application. Accordingly, processor scheduling has been a topic of great interest to the Archons project for many years. The concept of *time-value functions* has been developed to describe the time constraints of activities in real-time systems. Time-value functions were originally developed by E. Douglas Jensen in the context of ballistic missile defense [Jensen 75]. The concept was subsequently developed by the Archons Project students and staff at CMU, and was incorporated in the Alpha operating system [Northcutt 88a].

As a result of the previous work in this area, it has become clear that the algorithms needed to perform time-driven resource management are computationally demanding. In addition, a distributed real-time system must also be capable of dealing with the exceptions and unexpected events that may occur as a result of the application environment. Because of its great importance and demanding requirements, the Alpha scheduling subsystem has become the center of great attention and much effort has been directed towards it in the course of developing practical solutions to the problem of time-driven resource management. In Alpha, the scheduling subsystem has been carefully partitioned from the rest of the system and a separate processing element is provided for it. Furthermore, the scheduling subsystem was designed to allow the simple and straightforward substitution of different scheduling algorithms, for the purpose of evaluation and comparison.

1.1.1 Real-Time Scheduling Requirements

In real-time systems, the timeliness of a computation is as much a part of correctness as is calculating the correct value. Threats to human life and property are the expected results of failure to meet either criteria. Most current approaches to meeting application time constraints involve the use of a sufficiently large amount of excess resources to ensure that the timeliness needs are met. This viewpoint is evident in the common belief that a real-time operating system is one that has fast context switching times and rapid interrupt handling.

Under this definition, any operating system running on a fast enough processor would be real-time. What is really needed, however, are scheduling techniques that make the best possible use of processor cycles, both when there are enough resources to fully satisfy the activities in the system, and when there are not enough cycles to meet all of an application time constraints.

Some current approaches to resource management, spawned from the excess resources school, involve the use of "guarantees" about resource availability. This brand of management can be the source of a great many problems, and is in itself antithetical to providing the best use of the resources in a system. It is impossible to make any guarantees about resource availability unless the system designer is willing to permit extremely urgent exception conditions to be ignored in favor of the less important ones which have been guaranteed the resources needed. Adding more resources to the system merely postpones the inevitable moment when even they will not be sufficient.

Responsiveness to time constraints should be based not on guarantees but on system software that does the best possible job (according to some meaningful application-level definition) at any given moment. This includes not only times when there are sufficient resources, but also times when unexpected events occur and there are not enough resources to handle every need of the application. Because many real-time systems are based on the belief that is possible to make guarantees about resource availability, they do not even address behavior in overload. As a result, when overloads inevitably occur, they are handled poorly. One form of good performance in overload is if the system's performance degrades proportionally to the amount of overload. Most existing systems experience complete failure when overloads occur. If the system is designed to have so much computation power that this will almost never happen, it could represent a serious waste of resources in the normal case. If, on the other hand, the system is designed for lower performance and does not have sufficient computational resources, system failure will occur. The optimum solution seems to be that the system should have sufficient computational resources to perform the absolutely vital system functions in the worst case. This way, the system does not fail in overload, but instead postpones or does not perform the activities of lesser importance in favor of those that are vital to successful mission completion.

1.1.2 Distribution Requirements

To be most effective, a mission-oriented distributed system must be strongly connected, and must perform its activities as a whole in a cooperative (as opposed to competitive) fashion. Alpha is the first distributed operating system to be truly *decentralized*. There is no central entity whose failure would doom the system. The Alpha system provides support for cooperation on a peer level which is implemented so that it can be logically viewed by the programmer as having all the behavioral characteristics (in terms of synchronization and coherency) that a centralized system provides.

The decentralized nature of Alpha has several important consequences with respect to scheduling the application processors. The primary example involves time-constrained computations that span multiple nodes. Proper handling of these distributed computations re-

quires that the scheduling subsystem must be made aware of the time constraints associated with a thread when it arrives at a node. If, when a thread spans a series of nodes, a failure occurs in one of the nodes in the chain, the computation performed by that node is lost and the computations based on its results are invalidated. Therefore, the head of the thread must move back to the point prior to the first failure and Alpha must intelligently schedule cleanup activities for the orphaned threads. This includes removing time constraints that were acquired on or after the broken node.

1.2 Testing Alpha Scheduling Policies

The importance of the scheduling function in the Alpha system demands that the policies used to allocate processing resources meet (as well as possible) the requirements of the application. It is of course impossible to know in advance what characteristics any given application may require of a scheduling policy. Certain characteristics have, in the project's experience, proven to be especially helpful in real-time command and control applications. Using these characteristics as a starting point, requirements for an application to run on the Alpha system to exercise each scheduling policy on each of these points were devised.

1.2.1 Test Objectives

The scheduling policies examined in this report vary widely in such characteristics as the amount and kind of information required, the computation time used in determining a schedule, and the criteria used to rank competing threads into a schedule. A number of aspects of each scheduling policy's behavior are of interest in this effort. The primary intent of this work was to evaluate the relative abilities of various scheduling policies when given a certain amount of information by the application. Also of interest is the examination and characterization of the behavior of each scheduling policy within the context of Alpha.

The policies tested here were evaluated under both underload and overload conditions. Because the policies each use a subset of the available time constraint information and because each policy makes a different use of this information, the behavior of an application can vary greatly from policy to policy and between underload and overload. It was of particular interest to discover how well the overload performance of the scheduling policies provided for graceful degradation of system function. In general, if a system is not overloaded, a scheduler should be able to create a schedule that permits all the activities in the system to complete before their critical times (the time at which completion of the activity would no longer result in any value to the system). On the other hand, there are scheduling policies that do not make use of all of the application-specified information that is available to the scheduling subsystem in Alpha, and cannot attain even this level of performance. Other policies make use of the full information available in the time-value functions, but use different criteria to determine schedules. The characterization of the resultant application-level behavior was of great interest in these experiments.

Also examined was the response of various scheduling policies to particular scenarios representing specific resource management problems. Furthermore, the sensitivity of scheduling policies to variations in their input parameters was also of interest, and experiments

were carried out to determine the extent to which the values of various parameters affect the schedules generated by each policy. These experiments provide clues about how the scheduling policies respond to small variations in constraints and how each policy makes trade-offs when faced with difficult resource management decisions.

It was also intended that the results of this effort would include additional information about the effectiveness of Alpha's design and performance. One item of particular interest is the effectiveness of the separation of policy from mechanism within the scheduling subsystem. Alpha was designed with a scheduling framework into which a wide range of user-specified scheduling policies can be inserted. This process of providing a collection of mechanisms and not mandating a specific policy is a characteristic of Alpha operating system as a whole.

An important objective of this work is the evaluation of scheduling policies to find ones which exhibit the desired behavior with respect to handling time constraints imposed by aperiodic events. It was hoped that the empirical examination of the behavior of different scheduling policies would suggest what which class of policies performs the best under conditions representative of real-time command and control applications.

1.2.2 Test Application Requirements

In order to meet these objectives, it was necessary to come up with a sample application which both contained the elements of a supervisory real-time control application and could illustrate in clear terms the efficacy of each scheduling policy with regards to its ability to successfully meet the given time constraints. The application must be one where it is possible to separate the effects of the scheduling policy from the other functions which affect the behavior of the application. To meet these two objectives, the application must contain certain elements. First of all, the sample application must contain hard time constraints—i.e. there must exist activities which demand response within a specific amount of time. If the application responds to a time-critical event after that time, the response will be of no benefit to the system. In most real-time systems, there are such activities which simply *must* be completed before a certain time has elapsed. However, there also exist applications where the penalty for not meeting a time constraint may not be the loss of the system, and in fact may not be very severe at all. Such activities are known as *soft time constraints*.

Another important feature required of the test application was the presence of multiple schedulable entities. It is necessary for the scheduler to have a reasonably large number of tasks contending for processor cycles. Having multiple concurrent activities helps ensure that the scheduler must constantly make scheduling decisions, and that the effects of those decisions will be visible at the application level. A small number of activities might reduce the demands on the system to a point where there was no longer any need for the scheduler to be intelligent about allocating processor cycles. Having multiple activities that each perform a similar function but with different time constraints would make it easier to see the impact of the time constraints clearly.

2. Scheduler Evaluation System

After considerable thought, an application was devised which meets most of the criteria expressed in the previous chapter. The abstract problem chosen for these experiments is the task of keeping several bouncing balls in the air by means of moving motorized paddles. Each ball has a paddle assigned to catch it. The application program that executes on Alpha is responsible for controlling the motion of the paddles to ensure that balls are not dropped. The application program receives sensor data giving the position and velocity of the balls, and produces a series of actuator commands to move the paddles into position.

The experimental environment is composed of two major components—the *mechanical subsystem* that includes the physical environment in which the control activity exists (i.e., the balls, paddles, walls, ceiling, floor, gravity, etc.), and the *control subsystem* that includes the application program that performs the control functions, the Alpha operating system, and the Alpha testbed on which it all runs. Figure 1 illustrates the major components of the experimental environment.

In the work described here, the mechanical subsystem consists of an accurate simulation of the mechanics of the bouncing balls and movable paddles. The system's sensors and actuators are also simulated. In addition to greatly simplifying the system, simulation of the mechanical subsystem permits the rapid and accurate gathering of data, and permits the straightforward manipulation of interesting experimental variables.

There are a number of software components which comprise the experimental environment. First is the Alpha application program which meets the given problem definition—i.e., the program must attempt to keep balls airborne by moving their associated paddles (the Alpha scheduling policy under test is responsible for scheduling the threads involved in this activity). Secondly, there is the simulator which replaces the mechanical subsystem—i.e., the devices that the program is designed to control and the physical environment in which they exist. Finally, there is a human experimenter to manage the experiments (e.g., start and stop simulation runs, alter simulation parameters, etc.).

2.1 Application Program Structure

The activities which comprise the application program have a variety of timeliness requirements. The selection and ordering of the activities executed is controlled by the Alpha scheduler. This causes the effectiveness of the application program to depend entirely on the quality of the decisions made by the scheduler in Alpha. Because each of the paddles in the application is assigned to a single ball, there is a separate system activity (i.e., thread) assigned to control the movement of each paddle. Therefore, when a paddle moves, the observer can know that the system activity responsible for the movement of that particular paddle has been scheduled and run.

The time constraints associated with threads are distinguished from each other in three ways. The first way is through *deadline* information about the current task (in this applica-

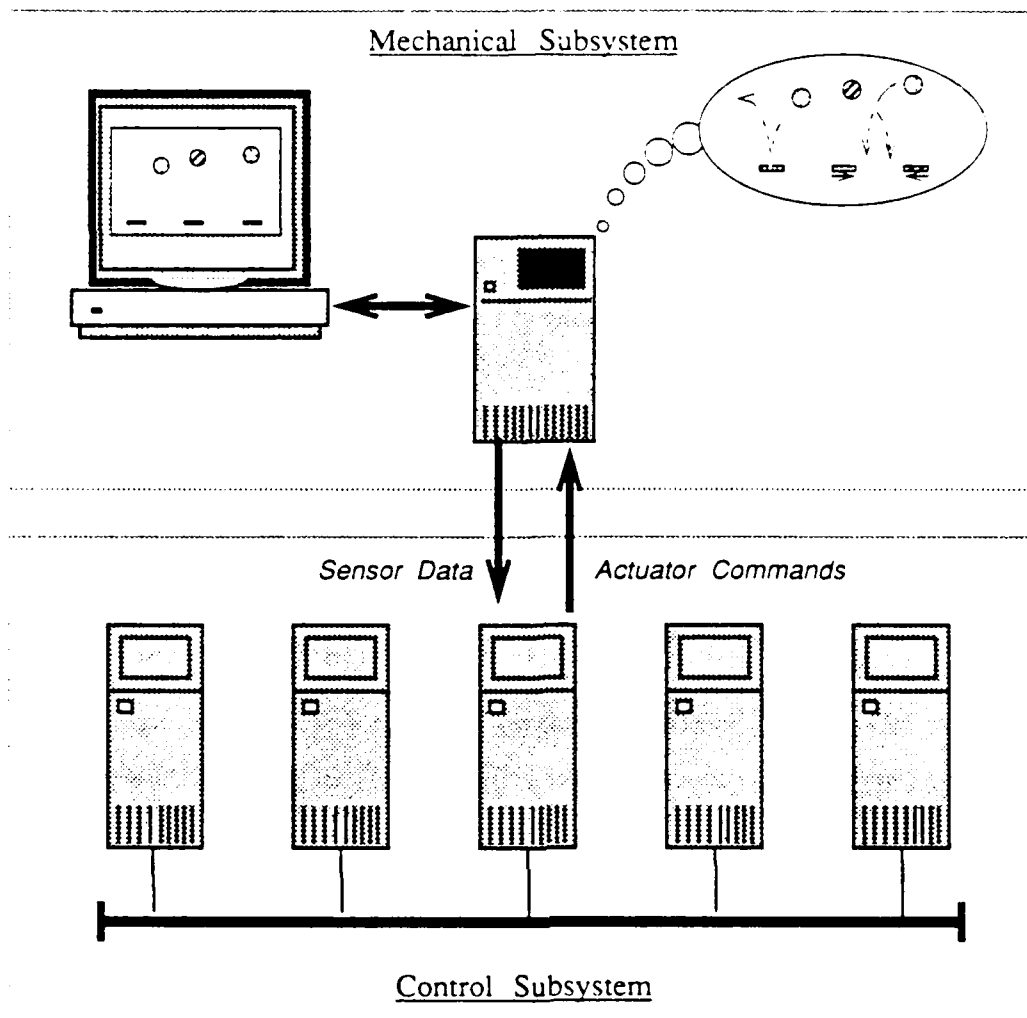


Figure 1: Experimental Environment

tion, the time until the assigned ball falls to the level of the paddle). The second way that time-critical activities are described is by their expected computation time. This estimate is the amount of processing resources expected to be required by a thread in order to meet a time constraint. Finally, there is an integer value that can be assigned by the experimenter to each thread. This value becomes a characteristic known as the *importance* of the thread. In these experiments it corresponds to the importance of the ball that the thread is attempting to keep aloft. These three characteristics of each paddle movement activity (i.e., thread time constraint) are provided by the application program to the Alpha scheduler. This time constraint information is provided so that thread execution schedules can be created based on the time a ball will take to fall to the floor, the time it will take to move its paddle to the ball intercept point, and the value of the ball to be caught. All three of these characteristics may be observed and controlled by the experimenter, so both the basis of the scheduler's decisions and the effects of the decisions themselves (i.e., the sequence of paddle movements) are directly visible.

The simplicity of the application limits the amount of application-level interference that could mask the effects of scheduling decisions. A single piece of application code can describe the correct operation every paddle activity in the system. It is only necessary to describe how, for a single paddle, to use the sensor information on the location of the ball to compute and execute a sequence of paddle movements that will place it in position to intercept the falling ball. The only difference between paddle control activities is the critical time, expected computation time, and importance information associated with each thread. The decisions about which balls to (attempt to) catch, and in what order, are as a result made entirely by the Alpha scheduler.

The simulator's interface to the application is provided through the Sun-UNIX remote procedure call mechanism. The simulator communicates with the application paddle management system to ensure that the tasks which move the paddles are running when the simulation is started, and to remove them when the simulation is stopped or the user removes the corresponding ball. To remove or add a paddle, the operator interface simply calls the corresponding remote procedure. These remote procedure calls are translated into Alpha invocations by the Alpha external communications interface. Requests for simulator operations begin as invocations from the application and arrive at the simulator as remote procedure calls. The translations between each of these systems is provided by interfacing software. Thus an invocation of a simulator interface operation eventually is performed as a remote procedure call on the Sun-UNIX system, and a remote procedure call of a paddle management routine becomes an invocation on an Alpha testbed node.

There are two major operations which tasks in the application can request from the simulator. These operations are GetSensor, which returns information concerning the current position and velocity of a given ball, and MovePaddle, which allows the application to move a paddle one increment to the left or right. MovePaddle requires a fixed amount of node computation time to run, as would be expected from an operation that required the continuous pulsing of a stepper motor. GetSensor requires some time for the data to arrive, so it involves blocking and waiting for the packet to arrive. Both operations were designed to model a physical system. There are also two major operations that the simulator can request the application to perform. These operations are AddPaddle and RemovePaddle. The application has a *paddle manager* which implements the paddle control operations that the simulator can invoke. The paddle manager allows the evaluation environment package to remotely create and delete threads that implement individual paddles.

The implementation of the paddle manager depends on the task management facilities available on the application system, as does the implementation of any auxiliary modules used to communicate with the evaluation environment's system. The design of the paddle tasks, however, is fairly straightforward and is mostly independent of the system on which they run. Each paddle task must query the position and velocity of its target ball, must calculate the predicted landing location, and must move the paddle to that location. Once the ball bounces or is dropped, the process is repeated. An important fact to realize is that the cyclic nature of this task does not make the application a *periodic* process in the sense of having predictable time constraints. The timeliness and processing requirements of the paddle tasks

may be different each time they repeat the sense-move loop. As in most supervisory real-time problems, the workload cannot be predicted in advance.

Two implementations of the application were developed. These were on a Sun UNIX system and the Alpha Release 1 testbed. The UNIX version was implemented to simplify testing the application interface to the simulator. The Alpha system was the object of the actual experiments.

2.1.1 UNIX Implementation

The structure of the UNIX implementation is a direct translation of the English description given above. The paddle tasks, which are implemented with UNIX processes, use the remote procedure call mechanism to access the simulator GetSensor and MovePaddle operations. Each paddle process calls GetSensor to obtain the position and velocity of the ball as well as the distance it can move in each paddle movement increment and the time each move takes. It then computes the distance that it must move to catch the ball at the intercept point, converts to the number of movement increments that it will require, and calls MovePaddle that many times, waiting the appropriate delay between each move. If the paddle arrives early, it uses the UNIX interval timer facility to block until the intercept occurs.

There were several unfortunate characteristics of this UNIX-based implementation. Probably the major problem was that UNIX, not being a real-time system, had no facilities for describing the time constraints of a task. Each paddle task had a time constraint which was the time required for a ball to fall to the level of the paddle, after which there was no use in continuing to move to the intercept point. There was no way to describe this time constraint to the system, and there was no way to tell the system that such an movement attempt should be aborted after it had already failed. It was possible to use the interval timer facility and a variety of complicated tests to generate the desired behavior. However, there was no general, natural way of describing the characteristics of the paddle tasks and having them considered as part of the scheduling process.

2.1.2 Alpha Implementation

Because there are general mechanisms for describing time constraints in the Alpha system, it was straightforward to implement the paddle tasks in a way that permitted them to describe their time constraints to the scheduling subsystem. The structure of the paddle task remains much the same from the UNIX version. However, the paddle task now used the information acquired from the GetSensor call to determine the time required to move the paddle to the intercept point. The importance of the task for the purposes of the scheduler is given by the value of the ball scaled by a constant. The value function is given by a constant which drops to zero at the time that the ball will pass the level of the paddle. Such a time constraint forms what is called a *hard deadline*.

Depending on the scheduling policy used, some or all of this time constraint information may be utilized to determine a good schedule. Under policies that support dynamic deadlines, the tasks can be automatically aborted out of their deadlines once the deadline has

passed (this eliminated the elaborate checking to determine if the ball has passed yet). As soon as the ball is dropped, the deadline is aborted.

2.2 External Environment Simulator

The external environment simulator serves as a surrogate for the real-world sensors and actuators comprising the external environment of a supervisory control system. It provides functions to read the sensors and move the actuators, and keeps track of the changes in the simulated environment. A graphical operator interface allows the user to control the simulation. These two modules together allow the experimenter to manipulate the situation the workload and record the results.

2.2.1 Simulator Structure

The simulator models the motion of the balls and the paddles and provides sensor information about the current state of both. The interface between the evaluation environment and the application was designed so that the application could be placed into a real, physical situation identical to that which was being simulated, and no change to the application would be necessary. In addition to modeling the motion of the balls, the simulator accepts requests from the application that cause the paddles to move. It also accepts commands from the operator interface that allow the user to start and stop the simulation, create or modify balls, and extract performance information as the application is running. Other actions performed by the simulator include notifying the application when a ball has been dropped.

The simulator performs most of its work during the simulation update period. This occurs once every time period, and involves updating the positions and velocities of all the presently existing balls, as well as checking for collisions and performing bounce calculations. These computations are performed by modeling elastic collisions with the walls and paddles. To better simulate the kinds of aperiodic loads which actual supervisory real-time systems experience, the bounce direction from a collision with the paddle is selected randomly. If during the process of updating the simulator state, it is discovered that any balls have been dropped, the application is notified and directed to remove the associated paddle. However, to enable the user to sustain a constant load on the system, it is possible, using the operator interface, to select certain balls that will remain in the simulation when dropped and will bounce back up above the paddle instead of being removed from the simulation. The output of the simulator includes timestamped messages indicating when balls are caught or dropped, and what the value of each ball is. With this information it is possible to compute a wide variety of useful statistics about the performance of the system.

2.2.2 Operator Interface

The operator interface provides the experimenter with control over the simulation. It displays a window that contains a small control panel for managing the simulation and a larger subwindow that shows the current simulation state (see Figure 2). There are buttons on the control panel that permit the user to stop and start the simulation. When in the stopped state, no paddle tasks are running on the Alpha system and the user may add or remove

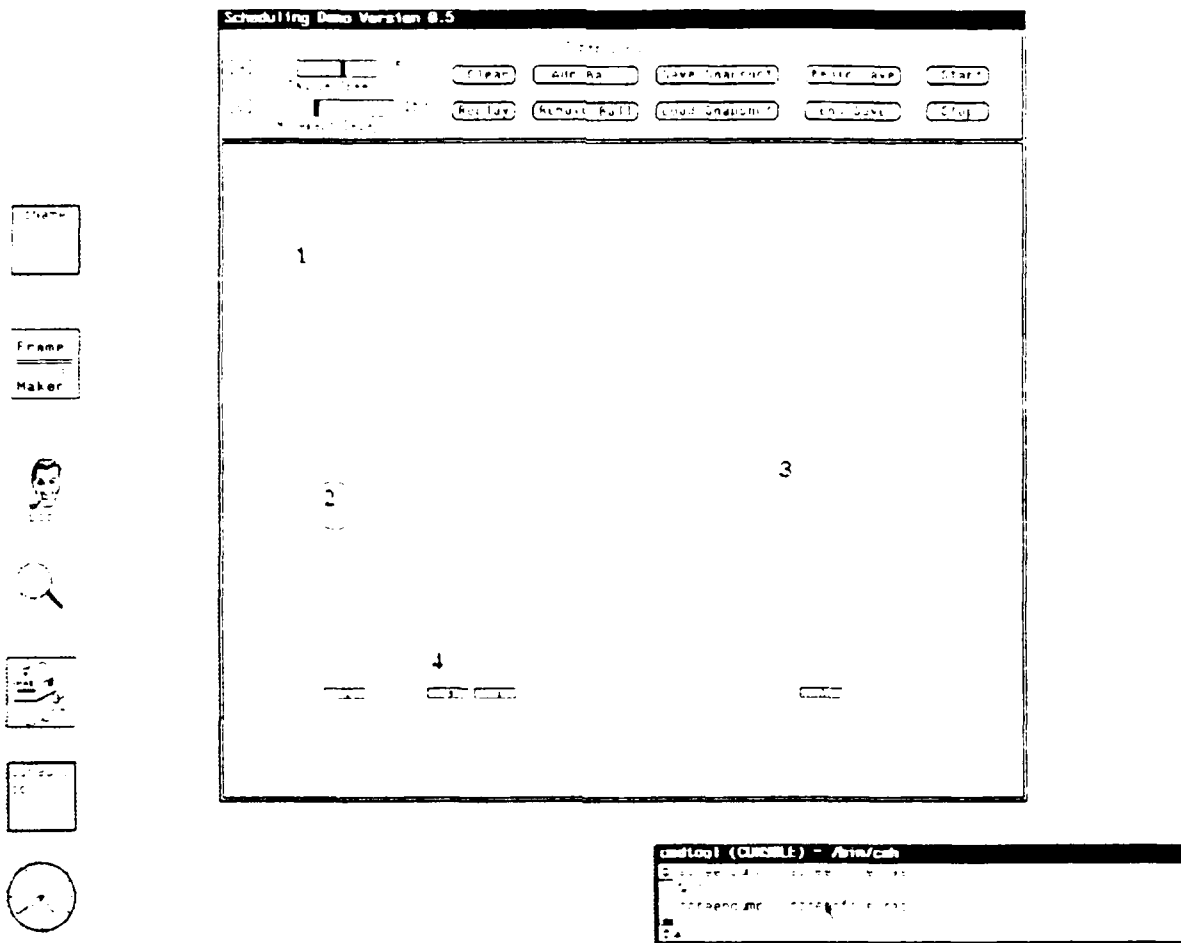


Figure 2: User Interface

balls and paddles without difficulty. Balls are added or removed by using two buttons reserved for that purpose. The operator selects locations by "clicking" the mouse button in the display area to indicate the ball or the position desired. Adding or removing a ball will result in the addition or removal of a corresponding paddle. In the stopped state, however, no paddles are shown; the paddles first appear when the simulation is started. There are also facilities for capturing and replaying sequences of action that appear in the display window, as well as for saving and restoring initial ball placements. Selecting a ball displays a panel that allows the user to alter its velocity, height, value, and paddle position.

When the user "clicks" the start button, the simulation begins and the balls start moving according to their given initial conditions. Once the simulation is started the paddles start moving to catch the balls. The user may alter the distance a paddle travels with a single pulse of the stepper motor, the *paddle speed*, and how much compute time each pulse takes, the *movement delay*. With these two controls, it is possible to change the amount of load present on the system. For example, increasing paddle speed decreases the number of computations required. Increasing the motion delay increases the computation time required.

This ability to adjust the positions and velocities of the balls and paddles greatly simplifies the task of experimenting with a variety of loading conditions.

3. The Structure of an Alpha Scheduler

The scheduling function in Release 1 of Alpha is managed by an independent subsystem that executes on a separate processor. A formalized interface defines the messages passed between the Alpha kernel and the scheduler. To simplify the development of scheduling policies, a general framework for implementing policies has been defined. The framework handles the actual message generation and processing, leaving the individual policy to decide how to respond to various messages.

The scheduler framework dispatches each kind of message from the application processor to a policy-supplied handler. Examples of these messages include messages that indicate that a task should be added to or removed, messages that define how the time constraints of the currently running task have changed, and others which support the distributed nature of Alpha, including cleanup for certain aborted operations. Each scheduling policy may choose how to respond to these messages and how to utilize the information provided therein. All eight schedulers which were implemented within this framework required only subsets of the information provided by these messages, and none needed any structure which was not easily created within this framework.

3.1 Scheduler/Kernel Interface

The function of a scheduler in the Alpha system is to determine which of the currently ready threads to run. To make this decision, the scheduler may use the time constraint information provided to the scheduler by the application. This information consists of three parts: a *time-value function*, indicating the time-varying value of completing a task at a given time, an *expected computation time*, the cycle time required to complete the task, and an *importance*, a value used to scale the value function. A scheduling policy must handle changes in these thread-provided parameters and, if necessary, correctly reevaluate the currently active schedule.

In Release 1 of Alpha, the scheduling subsystem executes on a separate processor from the application (and most of the Alpha kernel). The scheduling framework implements a queued, message-based communication channel between the processors. Messages are transmitted in two parts. There is a command part, which indicates the command to be performed, and a body which gives parameters to the command. The commands are:

- **Add:** used to tell the scheduler of the presence of a thread which is newly available for scheduling. This can occur when a new thread is created, when a blocked thread is unblocked, or when a thread from another node starts running on the node. This last variant is referred to as a "surrogate add" because the structures representing the thread on this node are acting as surrogates for the thread structure on the node where it was created. There is also a version known as a *delayed add*; this provides a time delayed Add command and is utilized in implementing the kernel Sleep operation. The parameters describing the function include a unique identifier for the scheduler to refer to in communications with the application processor. If the Add is a surrogate add, i.e. if the thread was created on another node and its point of con-

rol has just arrived on this node, the parameters include information about the time constraints it accumulated prior to arrival on this node.

- **Remove:** indicates that the current thread has voluntarily given up the processor. This usually occurs when the thread which is currently running has blocked awaiting some activity. Another version called Kill is used when the currently running thread has permanently given up the processor. This occurs when a thread returns from the invocation it started in, and when a thread that originated on another processor completely finishes its work on the node.
- **Change:** This command arrives when the currently running thread has altered its time constraints in some way. There are three ways in which this can occur. First, the thread may enter a new deadline. This is referred to as *pushing time constraints*, because deadlines may be nested. Second, the thread may exit a deadline. For similar reasons, this is referred to as *popping time constraints*. Finally, the thread can change its importance.
- **Access Scheduling Information:** There are four commands of this type. They all relate to conditions when the time constraints must be accessed or altered outside of the normal stacking methods. The *dump* subcommand is used to gather a thread's time constraints in preparation for a remote invocation. The *update* command is used to update time constraint information such as total computation time after the thread has been executing on a remote node. There are two commands that deal with abort conditions that permit the correct time constraints for abort processing to be determined. These are the *Get* and *Set Abort Scheduling Information* commands. If a deadline aborts, the *Get* command is used to determine the correct time constraints to operate the abort cleanup code with. The *Set* command is used to force the time constraints of a specific thread into those which are required for its abort processing.
- **Preemption Confirmation:** This is a necessary element of the communication between scheduler and kernel. It is sent after a request to replace the currently running thread with another has been successfully processed. This allows the scheduler to update the amount of time a thread has run. There are two variations of this command, one of which indicates that the currently running thread was preempted successfully, and one which indicates that the processor was idle when the preemption was requested. This message is necessary to provide the scheduler with a consistent picture of what is happening on the application processor.
- **Statistics Control:** There are two of these commands, that turn on and off optional statistics gathering, if the current scheduling policy supports logging.

There are only a few circumstances in which the scheduler must initiate communications with the kernel. Several of these instances occur as part of handshaking involved with the scheduler commands. Two, however, are of fundamental importance to the operation of the system. The *Preempt* command, which has been previously discussed, is used to tell the kernel to run a specific thread. The *Abort* command is used when the scheduler determines that a thread's critical time has passed. It initiates abort processing for the missed deadline. This command is, of course, only of interest for policies that support deadlines.

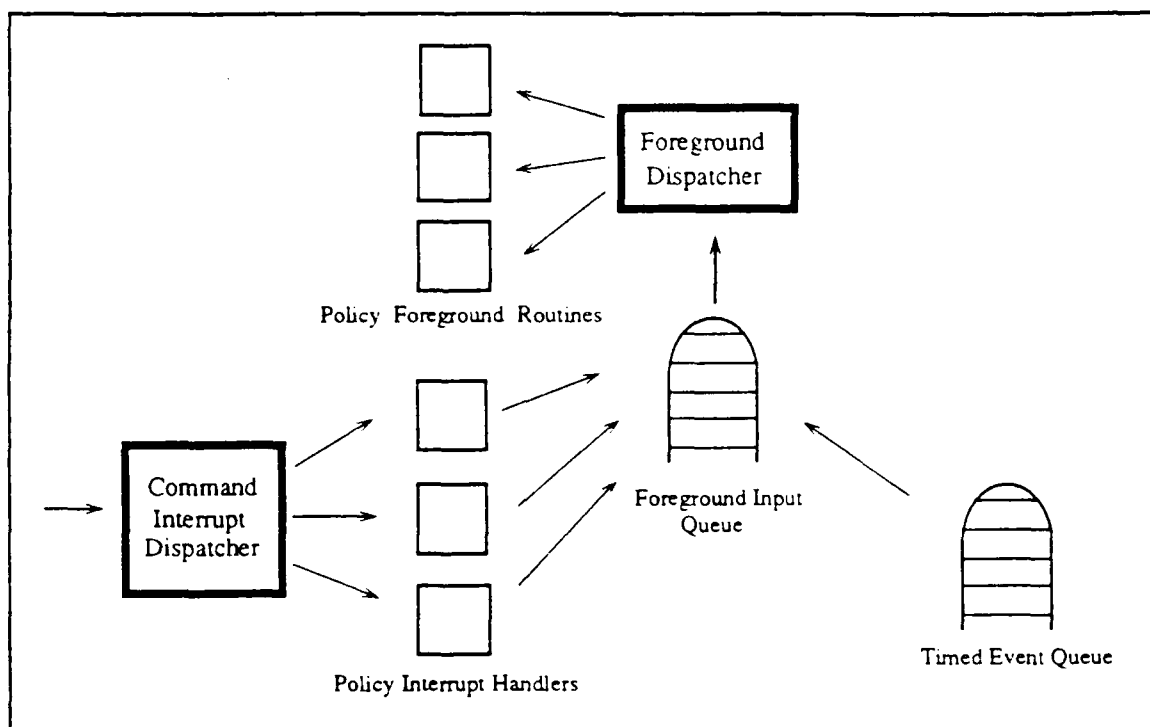


Figure 3: Basic Scheduler Structure

3.2 Structure of Generic Scheduler

Each of the scheduler commands described in the previous section must be implemented by every scheduling policy. The policy routines which implement these commands are automatically called when the commands are received. It is possible to supply a null function if the command is to be ignored by the policy, or an error function if the command should never be received by the policy.

Each policy must supply two types of command handlers. The first kind is called when the command is received under interrupt, and allows rapid handling of messages as well as general interrupt-level processing. In addition, the scheduler framework supplies an internal message queue which may be used to queue commands up for foreground processing (see Figure 3). The framework calls foreground procedures corresponding to the commands in the queue as they emerge. Any part of the handling for a command may be performed either under interrupt or in the foreground. If all the commands are to be handled in the foreground, then the interrupt tasks would simply enqueue the incoming commands into the foreground queue.

The minimum requirements for a scheduler in the Alpha system are fairly simple. The scheduler must remember the set of ready threads, and if the set is non-empty, make sure that there is always something running on the application processor. The scheduler framework provides a queue package which aids in the manipulation of schedules.

Scheduling policies that are more sophisticated, generally share some common characteristics. First, the schedulers generally construct a list describing the order in which threads should run. When a thread is added to the ready list or the time constraints of a thread are changed, some computation is required to determine whether or not to reorder the list. The operation of other commands, if any, depend on how the policy utilized the scheduling parameters provided by the threads.

4. Scheduling Policies

Five of the eight policies implemented for use in the Alpha system were selected for evaluation in these experiments. The following sections briefly describe the operation of each of these policies.

4.1 Round Robin

The Round Robin policy is the fairest of all the schedulers. It treats all threads in the system equally, giving each eligible thread the same fraction of the processor time.

The implementation of the Round Robin policy is straightforward. Every 100ms a timer event is inserted into the queue by the policy timer interrupt routine. Upon receipt of a timer event, the foreground command processor calls the timer routine, which removes the element at the head of the queue and inserts it at the end. If the new head is different from the old head, the application processor is preempted with the new head of the queue. Add and Remove commands are handled in the simplest possible way. If the command is an Add, the new thread is added at the end of the ready queue. If the command is a Remove, the thread is removed from the ready queue, and the application processor is preempted with the new head of the queue. Under Round Robin, time constraints are ignored, as are the miscellaneous commands associated with accessing scheduling information.

4.2 Static Priority

Static Priority scheduling is the simplest method of dealing with scheduling parameter differences between the threads available to run. Each thread is assigned a priority which is equal to the importance part of its time constraint. The scheduler always selects the thread with the highest priority, or one of them, if there are several with the same importance to run. The time-criticality of the thread not taken into account.

The implementation of Static Priority requires no timer usage. The scheduler framework's queue package includes ordered insertion routines, so the Add handler simply makes an insertion into the ready queue, then preempts the application processor if the head changes. The Remove handler removes the thread and preempts with the new head of the queue. The Change handler handles requests to change the importance of threads. This change is accomplished by removing the thread from the queue and reinserting it with the new importance. Again, if the head of the queue changes, the application processor is preempted with the new head of the queue.

4.3 Deadline

The Deadline policy always schedules the thread with the closest critical time. For this implementation, the Deadline policy was extended to provide for the abortion of deadlines that have expired. If a thread misses a deadline, it is forced out of the deadline section of code and begins executing a user-defined deadline abort handler. If the system has enough

cycles to meet all of the application deadlines, the Deadline policy is optimal—every time constraint is satisfied. If there exist deadlines which cannot be met, which is what is meant by *overload*, then the Deadline policy may perform very poorly. For example, this policy may attempt to run threads which cannot meet their deadlines, but which have early deadlines. The deadline policy is implemented as a priority scheme where the time to deadline forms the priority and the head of the queue is the item with the lowest priority. However, the Deadline policy uses the timer facilities to make sure each thread is aborted when its deadline passes.

4.4 Shortest Processing Time

The Shortest Processing Time policy considers only the required computation time for each thread. SPT always selects the thread with the lowest remaining computation time to run. It is similar to Deadline and Static Priority in that it uses only a single figure of merit to determine a schedule. It lacks optimal behavior of Deadline in underload, but may perform well under overload since it is more likely to pick threads which can be performed. SPT in general attempts to maximize the system throughput by completing as many threads as possible per unit time. SPT as implemented for Alpha has the automatic deadline abort mechanism mentioned above. The implementation of SPT is almost identical to that of Deadline, except the schedule queue is ordered by the remaining processing time.

4.5 Best-Effort

Given the problems with the other schedulers, which use only a small portion of the available information to construct a schedule, it is clear that any superior scheduling policies will have to make decisions based on all the information in the time constraints as well as the expected computation time. It is known that Deadline scheduling is optimal when the system is underloaded. An improved scheduler can therefore use a Deadline scheduling method when the system is underloaded. If the system is overloaded, the scheduler must decide which threads *not* to run. Threads that cannot complete their deadlines are one obvious choice. If that is not sufficient, the scheduler chooses those threads whose completion would be the least useful to the system. One metric of utility to the system is called the *value density* of a thread. The Archons project has developed a scheduling policy, called the Best-Effort policy, that uses value density as a metric when shedding load. Refer to [Locke 86] for a complete description of the Best-Effort algorithm.

The implementation of the Best-Effort policy maintains a queue of threads in deadline order. As each new thread is inserted, the policy checks to see if overload has been reached. When overload is reached, the policy sheds load by removing the least valuable threads until the system is no longer overloaded.

5. Experimental Results

This chapter describes the experiments performed on the chosen scheduling policies and their results. Justification is provided for the particular experiments that were performed, and the results of the experiments are presented.

5.1 Experimental Design

The goal of these experiments was to develop an understanding of the behavior and performance of the schedulers described in the preceding chapter. The experiments performed provide a broad range of quantitative information about each of the schedulers used. By varying the loading conditions and analyzing the resulting data, many different metrics could be extracted to compare the policies.

5.1.1 Load Generation

It is possible to increase or decrease the number of threads in the application, by changing the number of balls in the scenario. Having a large number of threads tends to make the tests more accurate since small variations in the behavior of the threads have less of an effect on the large-scale behavior of the application. With more threads it is easier to ensure that the scheduling policy being tested always has ready threads to choose from, thus improving the quality of the information extracted from the test.

The loading characteristics are also affected by the initial state of the balls and paddles. It is possible, but in general not desirable, to cause the conditions at the start of the simulation to be more or less favorable by placing the paddles farther away or closer to the balls they are catching, increasing the velocity of the balls, etc. This type of testing was done to analyze individual decisions made by each policy as an aid to understanding their behavior, but is detrimental to observing long-term effects since it introduces start-up transients that obscure the steady-state performance.

There are two ways to increase or decrease the computation time required by each thread in the system. The first method is to change the paddle speed. Increasing the paddle speed reduces the number of times each thread consumes a block of processor cycles. Decreasing the paddle speed has the opposite effect. The other way to change the computation time is to alter the movement delay. Increasing this parameter will result in increased computation by each thread since more effort is required to move the paddle each step on its way to the intercept point.

5.1.2 Evaluation Metrics

There are several factors which must be considered in order to make a fair comparison between scheduling policies. The different policies have implementations which may vary considerably from that which may be optimally attained. Thus, it is important to factor out the various problems that are due only to the implementation of the policy. The primary factor which affects the apparent performance of the schedulers is the fact that some schedulers

may take longer to provide a thread for the application to run after the currently running thread blocks. This effect is primarily realized in the system throughput, resulting in an apparently faster application. Part of the data collected is a measure of the system throughput, the total number of paddle movements accomplished in each minute. It is possible to utilize this throughput measure to normalize all of the schedulers to a common measure, which was selected to be the value to the system accomplished by the Round Robin scheduler. To determine the normalized value to the system obtained by a given scheduler, it is only necessary to divide by the throughput measure of the policy and multiply by the throughput measure of Round Robin. In other words, the data is interpreted as though it came from a scheduler which had the throughput of Round Robin, but made different decisions. This enables the examination of the *decisions* made by the scheduler independent of the quality of the policy implementation.

The time-value curve of a system activity is composed of information on how the completion value of a segment of code varies with time. In combination with the importance, it can provide the scheduler with knowledge of how much value the system would accrue from the computation if the activity was scheduled at a certain time. A scheduling policy should in theory be able to determine the most valuable schedule possible from this information. However, some policies use only subsets of this information, and use the information in different ways, and thus may create other schedules of varying worth. The aggregate value to the system created by each scheduling policy's allocation of computing cycles is the primary metric used to compare policies in this report. Thus the main criteria we will use as a metric to compare scheduling policies is the extent to which each policy maximizes the value provided to the system by the application. This is the logical basis on which to judge schedulers, since a scheduling policy is intended to translate as best as possible the characteristics of each thread into a good schedule for the system. Since the set of characteristics provided by threads in Alpha is the completion value as a function of time, then the ideal scheduler is one that selects the schedule that provides the greatest value. Over long periods of time, the better policies will provide more value to the system than inferior ones.

The second major metric used to compare policies is the percentage of the time constraints met by each policy. This metric is of secondary importance since maximizing the number of met time constraints does not necessarily maximize the value obtained for the system. Nevertheless, it may provide clues as to why different policies perform as they do.

5.2 Behavior Analysis

To understand why each scheduling policy behaves as it does, it is helpful to examine some specific scheduling scenarios and to analyze how each of the schedulers would respond.

The first scenario of interest is one where all the balls may be caught, but only if a certain sequence is followed. This condition is shown in Figure 4. In this picture, ball A takes longer to return to the intercept level than ball B. Static Priority will fail in this case since ball B bounces twice for every single bounce of ball A. If one were to assign a higher priority to ball A, ball A would always be caught first and ball B would be missed on its second bounce. If

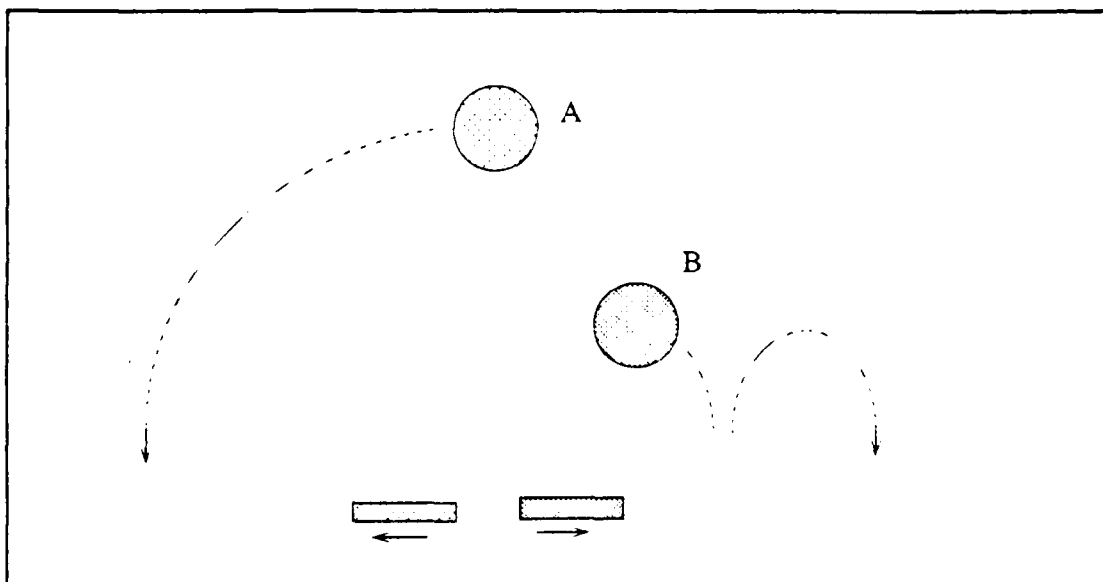


Figure 4: Dynamic Priority Example

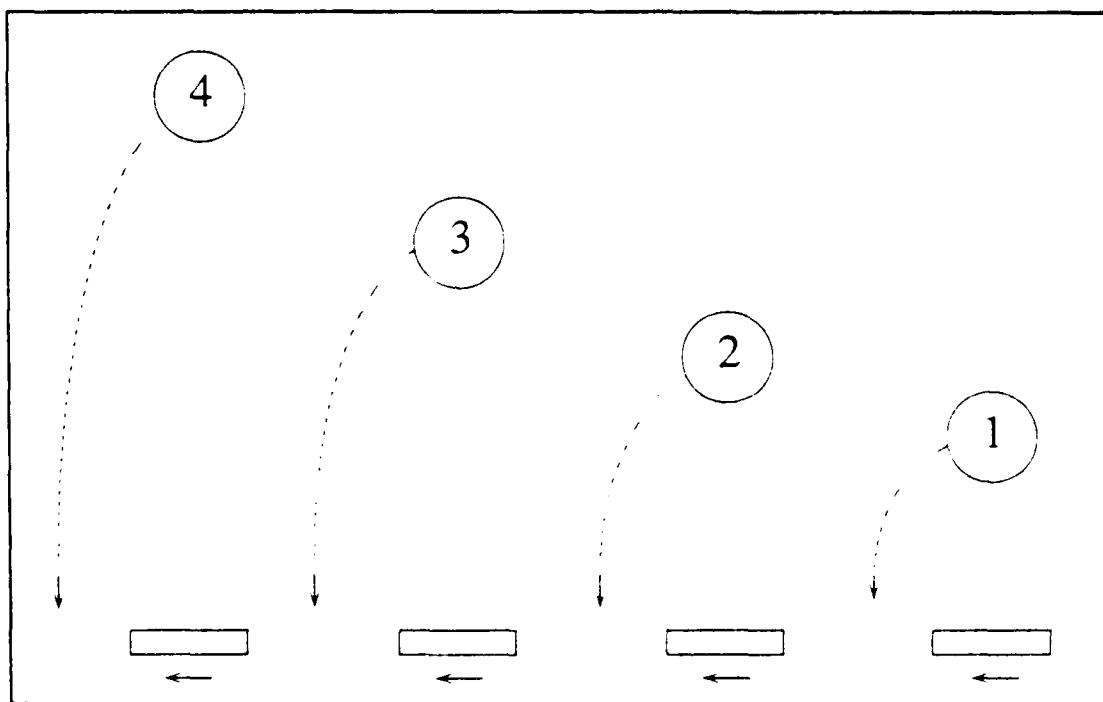


Figure 5: Inverted Deadline/Value Example

one were to assign a higher priority to ball B, A would be missed. Both Deadline and Best-Effort, on the other hand, would catch ball A, then ball B twice, then ball A again. SPT would give a higher priority to catching B, which takes less computation time, and so would work exactly like Static Priority when the higher priority is given to ball B.

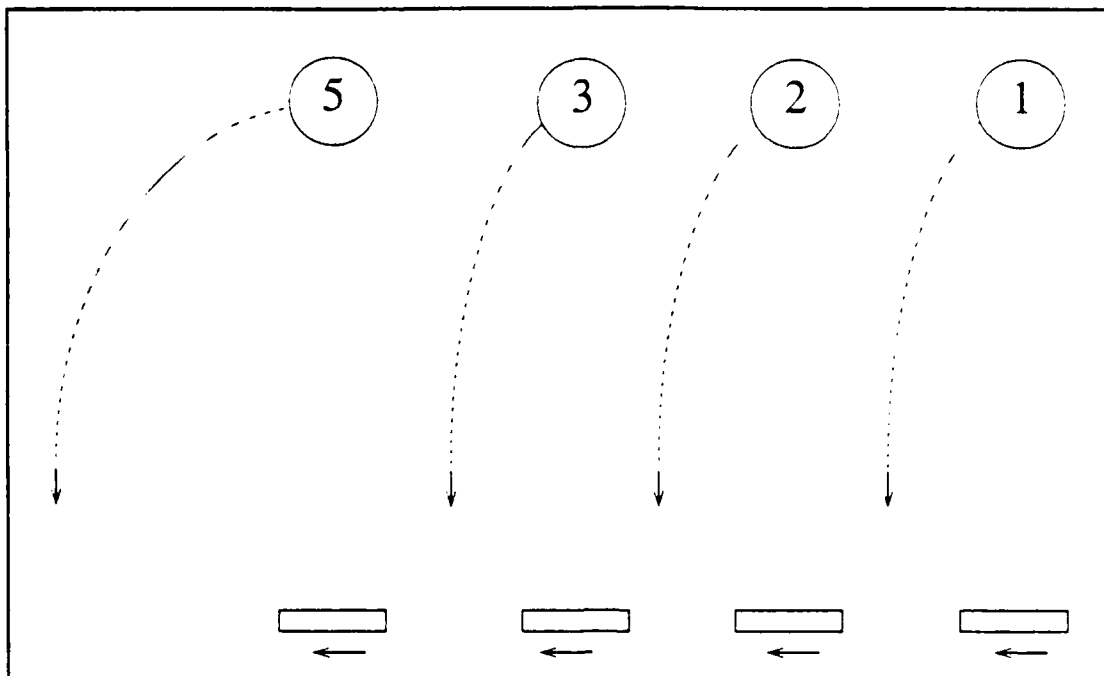


Figure 6: Simple Overload Example

Another underload scenario is shown in Figure 5. There are four balls with values 1 through 4. The value 4 ball is farthest from its intercept point and value 1 ball is the nearest. The paddle assigned to each ball is one time unit away from its intercept point and each ball is as many time units away from the intercept point as its value. In this case, the urgency of each computation is the inverse of the importance of the computation (i.e., the higher the value of the ball, the less urgent it is to catch it). Static Priority will fail badly on this scenario. It will first move to intercept the value 4 ball, thus dropping the value 1 ball. It will then move to intercept the value 3 ball and drop the value 2 ball. Static Priority will successfully catch the balls of value 3 and 4, but will drop those with value 1 and 2. Both Deadline and Best-Effort will catch all of the balls in this scenario, while SPT will act in random order (all the paddles take one time unit to move into position).

The next case of interest is one where it is not possible to catch all of the balls (Figure 6). The value 5 ball is three time units away from the intercept point, as is its paddle. The value 3, 2 and 1 balls are also each three time units away from their respective intercept points; but their paddles are each one time unit away from the intercept point. Static Priority would successfully catch the value 5 ball, dropping balls of value 3, 2, and 1 in the process. Best-Effort would recognize the overload situation and abort the catching of the value 5 ball in favor of catching the value 1, 2, and 3 balls that provide more value to the system. Deadline would catch two of the three smaller value balls, then select randomly between moving one time increment toward the value 5 ball (which now cannot be caught) or catching the third small ball. SPT would, in this case, successfully catch the three small balls, because they all require fewer computational resources to complete.

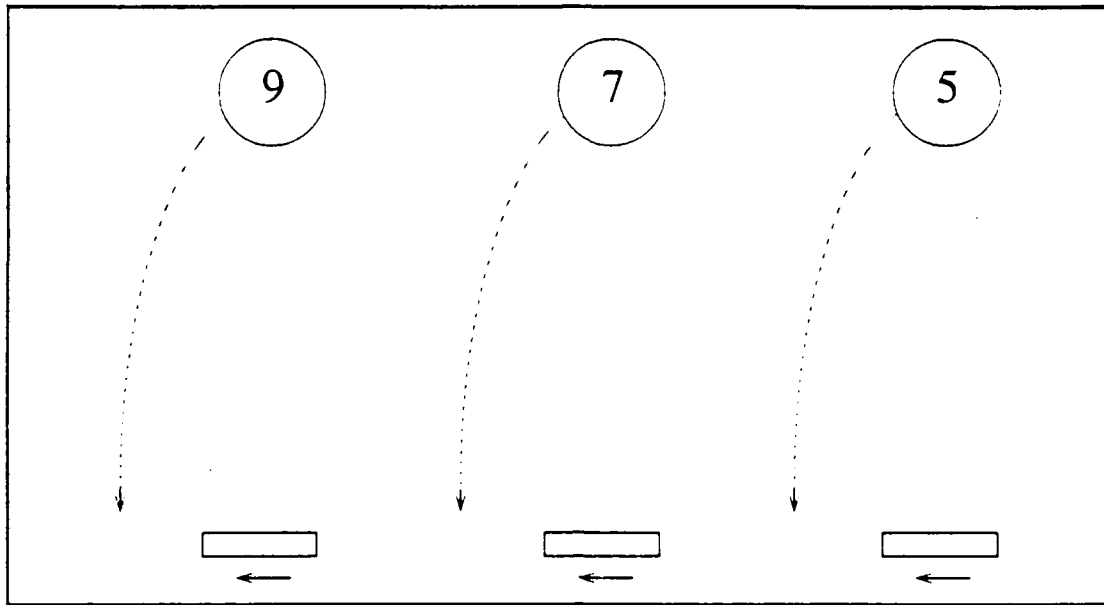


Figure 7: Value Selection in Overload

Another interesting scenario is shown in Figure 7. This figure represents another overload situation. All three balls will arrive at their intercept points at the same time (two time units). Each of the paddles is only one time unit away from their intercept points. Both Best-Effort and Static Priority will select the value 7 and 9 balls, while both SPT and Deadline will select randomly (all have the same deadline and computation time).

Finally, there is the case in which one ball is completely impossible to catch. Such a scenario is shown in Figure 8 where a ball of value 3 is one unit of time from its intercept point, while its paddle is two time units away. There is another ball of value 1 which is one unit of time away from the intercept point, as is its paddle. Static Priority will try to catch the uncatchable ball of value 3 while disregarding the catchable value 1 ball. Best-Effort will abort the paddle of the value 3 ball since its deadline cannot be made, and will catch the value 1 ball. Deadline will choose randomly between the two balls (which have the same deadlines). SPT will catch the ball of value 1 since it requires less computation time.

These scenarios indicate that the Best-Effort policy behaves well in a variety of different loading conditions. Unlike other policies which may perform well in a limited domain, the Best-Effort policy uses the full information available in the time-value function specifications to intelligently schedule tasks in both underload and overload situations.

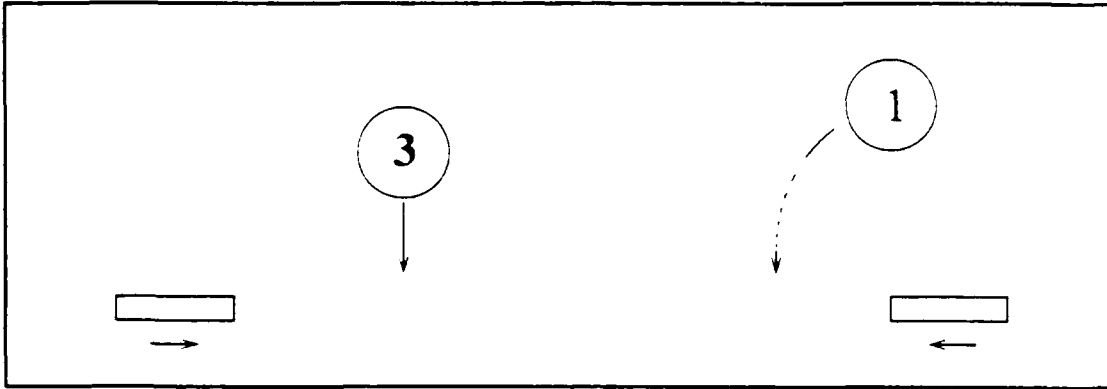


Figure 8: Impossible Time Constraint

5.3 Simulation Results

Several metrics for long-term performance were collected from each scheduler. The first measure examines how sensitive each scheduler was to variations in thread importance (i.e., ball value). The second metric records how successful each of the scheduling policies was at meeting the application time constraints (in terms of percentage of time constraints satisfied). The final indicator combines both ball value and time constraint measures to judge the overall performance of each policy.

5.3.1 Thread Importance Sensitivity

As may be seen from the graphs of percent caught versus ball value (shown on the following pages), each scheduler has a characteristic form indicating the trade-offs it makes when catching balls of different values. Round Robin, for example, has a flat graph (Figure 9) indicating that it does not distinguish between threads based on their value. Each thread in the system receives the same fraction of the available processor cycles. As expected, increasing the paddle speed increases the application's ability to catch balls. The Deadline algorithm (Figure 10) is also not responsive to ball value and shows similar behavior.

The Shortest Processing Time algorithm (Figure 11), which also disregards thread importance, shows some sharp differences between ball values. This effect is due to the particular way in which the application interacts with this scheduling policy. The amount of movement, and thus the compute time, needed to catch a given ball often increases after the ball is missed. The paddle needs to move farther to make the next catch, has a greater required computation time, will thus be less likely to be scheduled, and so also less likely to catch the ball. If, on the other hand, the paddle catches the ball, it will be closer to the intercept point for the next catch, will require less computation time, will be more likely to be scheduled, and will be more likely to again catch the ball. Thus, balls, once caught, are likely to continue being caught. Once dropped, they are likely to continue being dropped. Therefore each ball will tend to either be caught well or caught poorly for most of each run. This explains why the SPT algorithm shows such a wide variation in catching percentages with respect to ball value. Most applications do not exhibit this type of behavior.

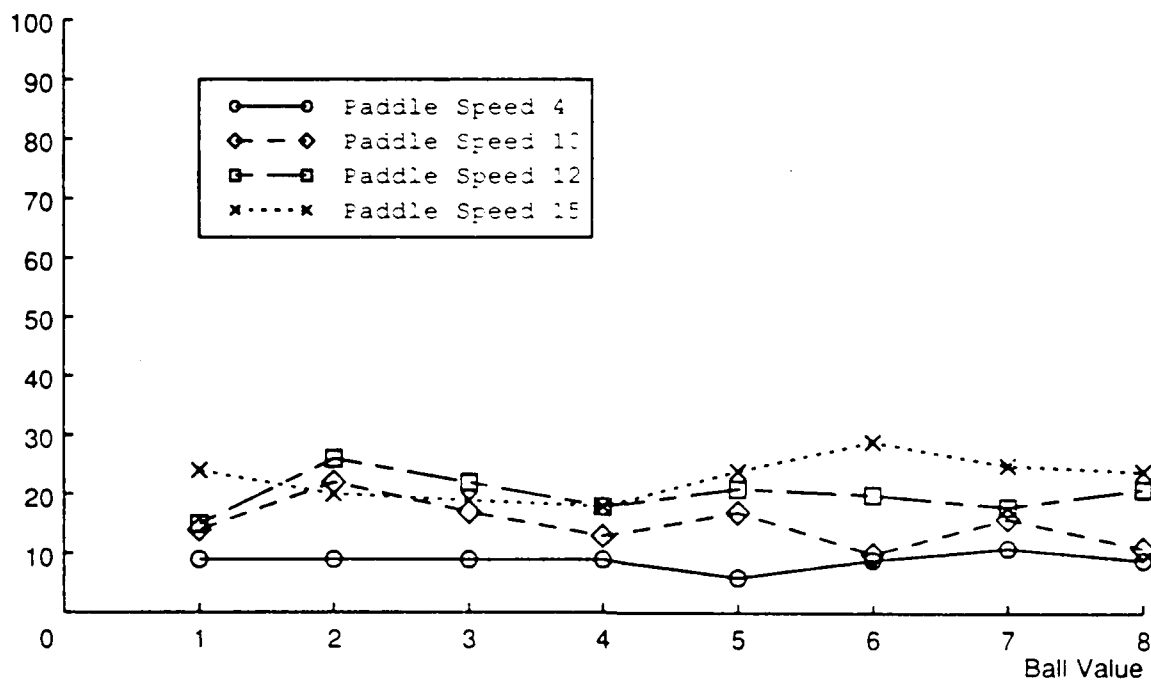


Figure 9: Round Robin Percent Caught

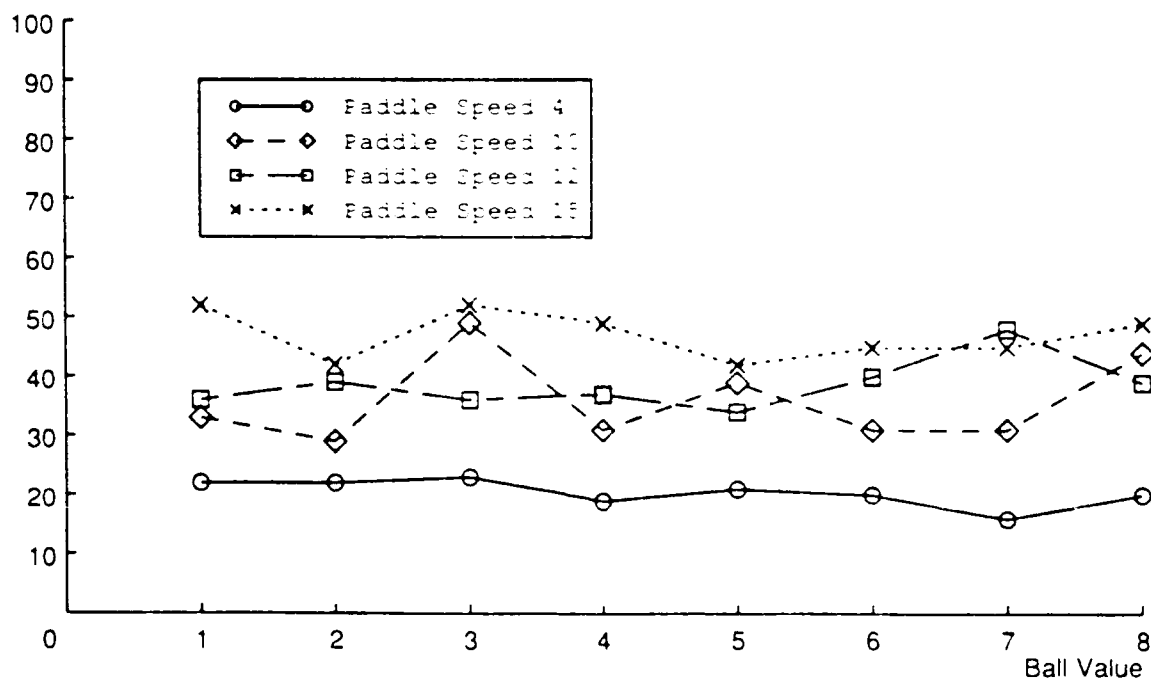


Figure 10: Deadline Percent Caught

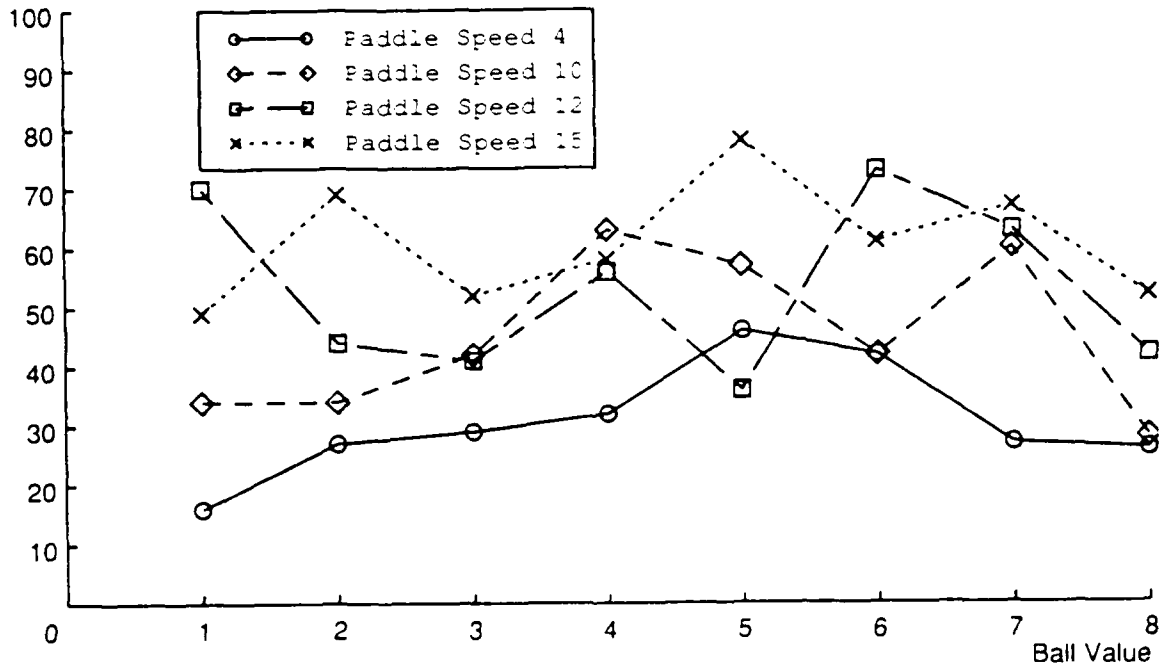


Figure 11: SPT Percent Caught

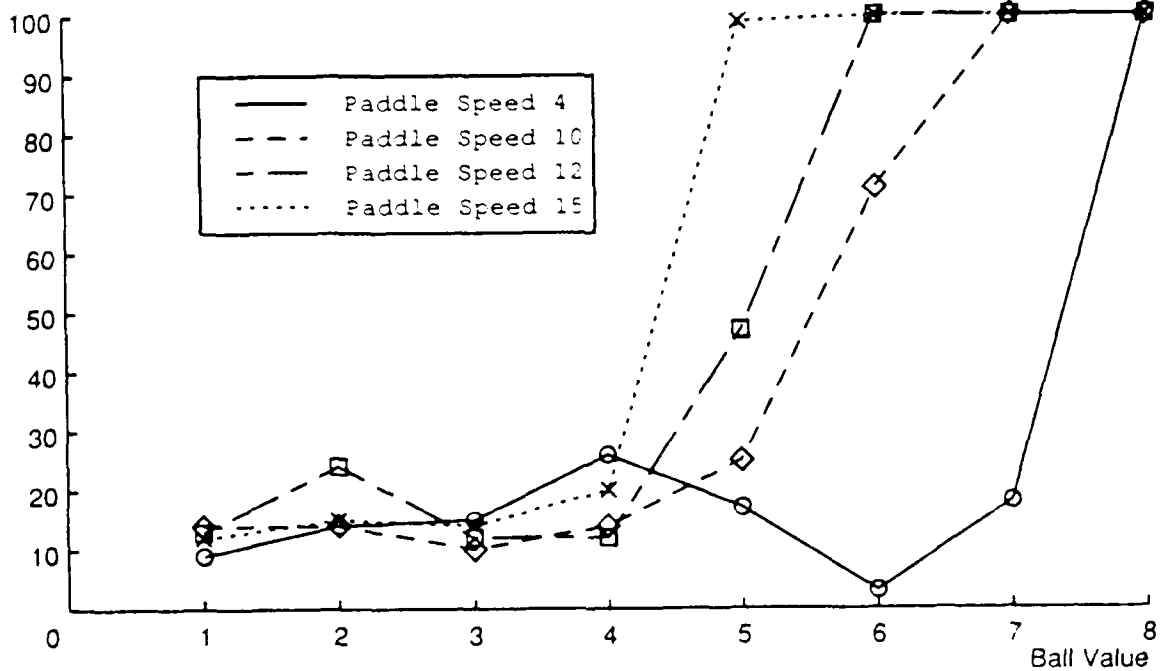


Figure 12: Static Priority Percent Caught

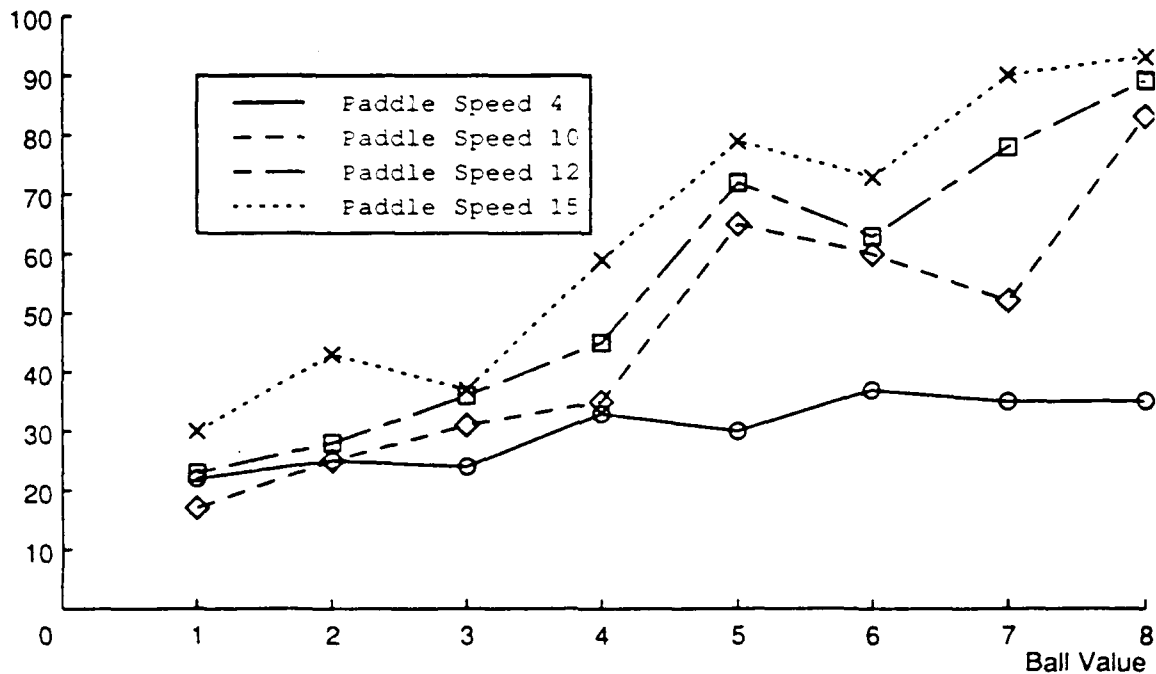


Figure 13: Best-Effort Percent Caught

Static Priority (Figure 12) has a very low catch rate for the small value balls, jumping to an almost perfect rate balls of higher values. Static Priority scheduling always runs the most important threads first. It therefore catches the high value balls (if it can), then attempts to catch the next highest value balls, etc. Increasing the paddle speed enables the application to catch more of the top value balls.

The characteristic curve for the Best-Effort policy (Figure 13) is smoother than those for the other policies. Like Static Priority, Best-Effort has a very good catching rate for the highest value balls. However, the rate of decrease between the high value balls and the lower value balls is more continuous. This behavior is a very important characteristic of the Best-Effort scheduler. Instead of concentrating only on the highest value balls, it tries to catch the combination of balls that maximizes the total value to the system. It may often be possible to schedule the successful capture of several balls of low value in place of a single one of higher value, and as a result provide better total value to the system. Compared to Static Priority's insistence on catching high valued balls to the exclusion of others, this selection can result in a considerable increase in the value obtained.

5.3.2 Meeting Application Time Constraints

One important method of comparing the schedulers is to examine how well each of them meets the thread deadlines, i.e. what fraction of the balls are successfully caught. The ratio of balls caught to the total number of potential catches is shown for each scheduler as a function of paddle speed in Figure 14. As expected, the ability of any scheduler to meet deadlines, and thus to catch balls, increases with paddle speed. It is interesting to note that the

Deadline policy fares very poorly on this experiment. Since the system is heavily overloaded (all percentages are $<100\%$), Deadline scheduling may waste significant time attempting to meet impossible time constraints. The SPT policy performs well since it concentrates on tasks which are easily accomplished. The Best-Effort policy performs as well as or better than any of the other policies because of its intelligent overload handling.

5.3.3 Maximizing Application Value

The total value metric is determined by dividing the value of the balls caught by the value that would have been obtained if all of the balls were always caught. Figure 15 illustrates the performance of each scheduling policy using this measure.

It is worthwhile to compare Figure 14 and Figure 15 (the two graphs are extracted from the same experimental data). The scheduling policies that ignore thread importance—Round Robin, Deadline, and SPT—catch almost the same percentage of balls as they do of value (their curves are practically identical in the two graphs). The two policies that consider thread importance—Static Priority and Best-Effort—catch more valuable balls and are shifted up by approximately 10% in the graph that indicates the percentage of the total value caught (Figure 15).

The policies that performed the worst using the value metric, Round Robin and Deadline, are also the schedulers that did the worst job of meeting time constraints. Clearly they achieve less value to the system simply because they catch fewer balls. The Shortest Processing Time scheduler achieves the next highest value to the system. It performs better than Round Robin and Deadline because it catches more balls, but it does worse than Static Priority and Best-Effort because it ignores value. Static Priority and Best-Effort are the most successful and are similar in performance for this particular set of experiments.

Static Priority and Best-Effort, are worthy of further consideration. One reason they perform well is that, unlike the other schedulers, value information plays an important part in scheduling decisions. One might wonder why Static Priority, which does not consider time constraints, is so successful at accruing value? The answer is found by examining how Static Priority distributes cycles to an arbitrary mix of threads. In overload, only the highest importance threads run (excluding all others) regardless of how likely their time constraints are to be met. In the short term, this may result in more dropped balls. If a thread's ability to meet an individual time constraint were independent of its past history, this behavior would result in a poor long-term performance as well. However, if expending cycles on a time constraint, even if the constraint is not satisfied, serves to improve the chances of meeting the next time constraint a thread establishes, then the cycles are not wasted and may benefit the system in the long term. This effect manifests itself in this application. If a scheduler concentrates solely on the highest value balls, some balls that could have been caught may be dropped. However, because they are receiving the majority of the processing cycles, the high value balls will tend to be the ones which are the easiest to catch. As a result, scheduling decisions which appear to be bad based on the available information may be good in the long term. Some applications do exhibit this type of "feedback;" however, the majority of tasks have a greater independence between activations.

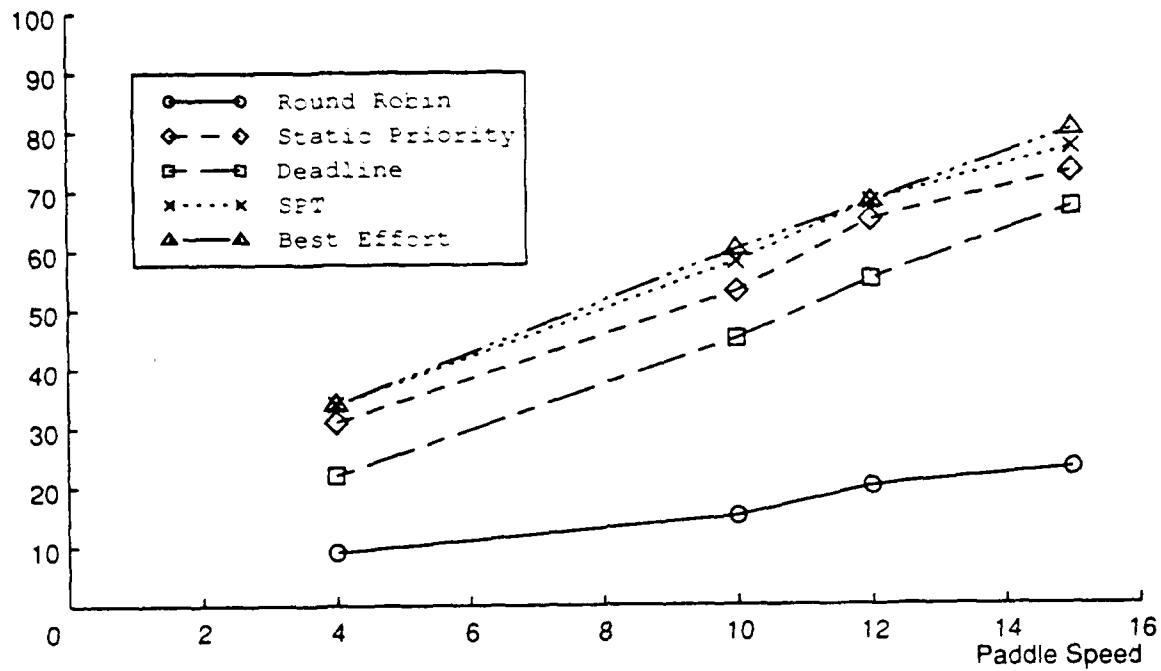


Figure 14: Percent of Balls Caught

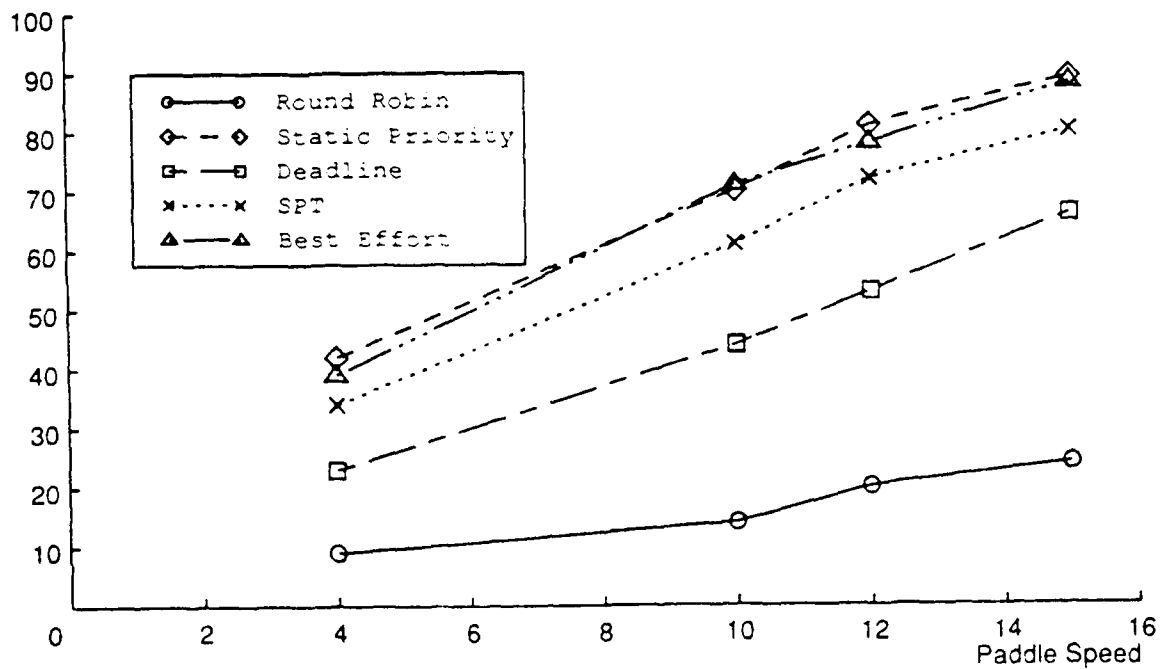


Figure 15: Percent of Value Caught

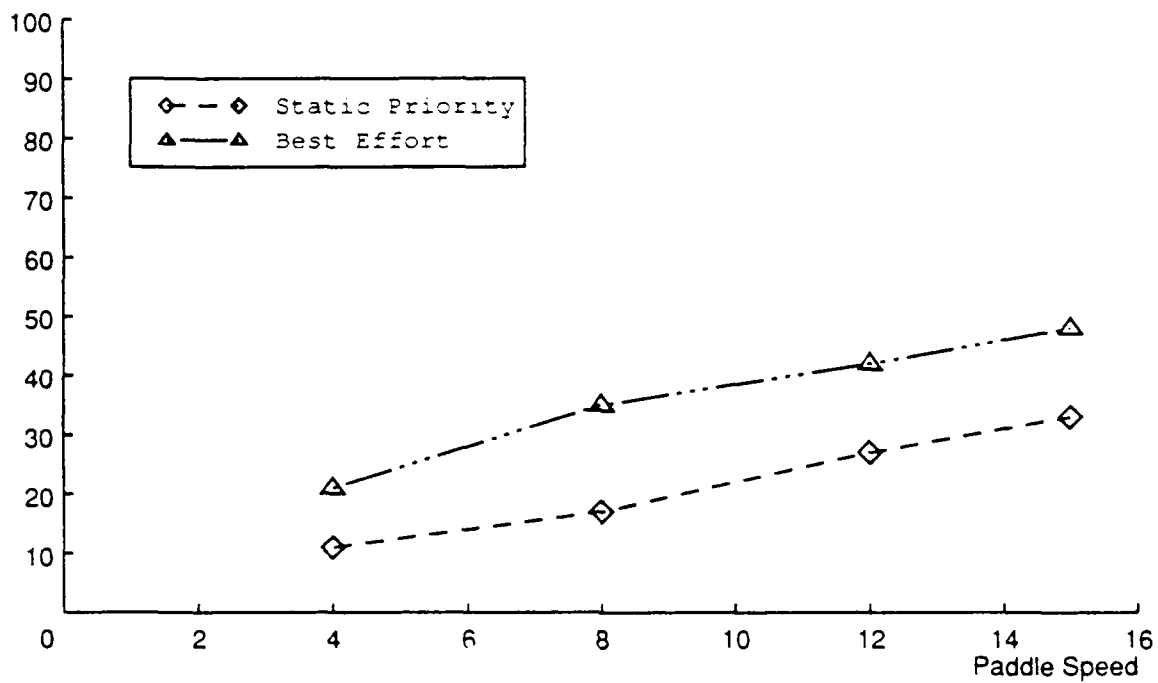


Figure 16: Percent of Value Caught with Random Replacement

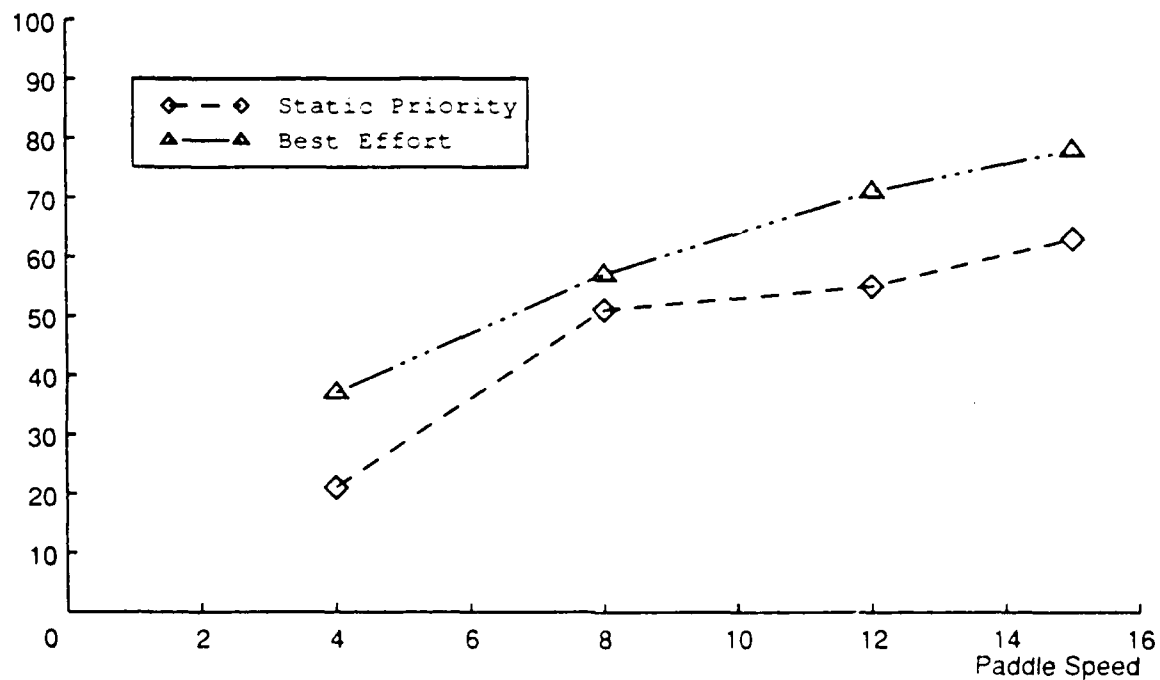


Figure 17: Percent of Value Caught with Smaller Value Variance

To study the behavior of independent tasks, a second series of experiments was performed. The application was altered so that after balls are caught or dropped they move to a random point in the air and begin falling again. Therefore, cycles spent on an unsuccessful catch is time poorly spent. A bad short-term decision cannot in general be converted into a good long-term one. The results of this modification are shown in Figure 16.

Another point to consider is whether the choice of values for the eight balls has an effect on the performance of the two schedulers that consider thread importance. The original set of eight balls valued one through eight was replaced by three balls of value six and five of value five. The results are shown in Figure 17. The comparative performance of the Best-Effort policy improves with this change in value distribution. The key to understanding this relative improvement in Best-Effort's performance is the fact that the balls with the highest value are no longer so useful to the system that focusing on them to the exclusion of others is profitable.

6. Conclusions

The best scheduler for a particular application would often be one designed precisely to match the requirements of the system. Such a scheduler could have complete knowledge of the application and could exploit application-specific information to achieve the best possible performance. For certain low-level systems where there are few different types of activities or where the events occurring in the system are completely predictable, custom schedulers may be feasible. As the variety of time constraints and frequency of unexpected events increases, however, it becomes more and more difficult to construct an application-specific scheduler that will operate correctly.

Since building a custom scheduler for each application is impractical, scheduling policies have been developed that utilize application-specified hints or requirements information to schedule activities in the way that will most benefit the application. In general, the more information given to the scheduler, the better the scheduling decisions can be. One consequence of using additional information is that scheduling decisions may become more complex. It is therefore necessary to balance the complexity of the scheduling operations with the resulting improvement in the application performance.

In the scheduling policies tested in this work, the amount of application-specified information used to make scheduling decisions varied from none (Round Robin) to a complete time-value function (Best-Effort). Predictably enough, it was the scheduler that used the most information about the application, Best-Effort, that showed the best behavior. The tests indicate that the extra computation time required in the Best-Effort algorithm was small enough to make the Best-Effort scheduling policy a good balance between performance and computational complexity.

References

- [Jensen 75] Jensen, E. D.
Time-Value Functions for BMD Radar Scheduling.
Technical Report, Honeywell System and Research Center, June 1975.
- [Locke 86] Locke, C. D.
Best-Effort Decision Making for Real-Time Scheduling.
Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, May 1986.
- [Northcutt 87] Northcutt, J. D.
Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel.
Academic Press, Boston, 1987.
- [Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Requirements and Rationale.
Archons Project Technical Report #88011, Department of Computer Science, Carnegie-Mellon University, January 1988.
- [Northcutt 88b] Northcutt, J. D. and Clark, R. K.
The Alpha Operating System: Programming Model.
Archons Project Technical Report #88021, Department of Computer Science, Carnegie Mellon University, February, 1988.

Time-Driven Scheduling of Composite Real-Time Activities

David P. Maynard

*Department of Electrical and Computer Engineering
Carnegie Mellon University*

November 1, 1989

Table of Contents

1.	Introduction.....	B-1
1.1	Application Domain.....	B-1
1.2	Related Work	B-3
1.3	Technical Approach.....	B-5
2.	Computational Model.....	B-6
2.1	Modeling Timeliness Requirements	B-6
2.2	Modeling Composite Activities	B-8
3.	Scheduling Techniques	B-10
4.	Evaluation Methodology	B-12
4.1	Evaluation Metrics	B-12
4.2	Workload Generation.....	B-13
5.	Research Schedule	B-15
6.	Research Contribution	B-16
	Bibliography	B-17

1. Introduction

Previous real-time scheduling research has primarily addressed simple tasks and has not considered that time-constrained activities may span multiple nodes in a distributed system and may have multiple, nested timeliness requirements. This research will investigate how limited additional information (relative to current algorithms) about the execution requirements of these composite real-time activities can be used to improve the quality of scheduling decisions made at each node in a distributed system.

1.1 Application Domain

There are several classes of real-time systems [Bennett 88]. This work considers a class known as *supervisory real-time* systems. Typical applications of this type include industrial factory automation (e.g., automobile manufacturing), platform management (e.g., space stations), and military command and control (e.g., air defense). These systems must operate correctly in highly dynamic environments where requirements and resources may vary gradually or may change suddenly without warning. Often, the applications execute on distributed computer systems where processing nodes are physically separated to reflect the structure of the problem or to enhance availability and survivability.

Supervisory real-time systems differ from low-level sampled data monitoring and control systems in several significant ways. Unlike low-level systems which consist primarily of simple periodic tasks, supervisory real-time systems manage a wide range of complex activities. These activities are characterized by the following features:

- Activities have stochastic arrival and execution times. It is often difficult or impossible to predict when or how often activities will be initiated.
- Activities may have a variety of critical timeliness requirements including hard deadlines, which indicate that an activity must be completed within a specific time interval for its result to be useful, and "softer" time constraints, which describe activities for which the value of completing the work varies across time.
- Activities are often composed of several execution *stages* which may involve computation on several different nodes in the system.
- Individual activities may have multiple nested timeliness requirements imposed by different levels of the application environment.

We call this class of activities which may have multiple execution stages and nested timeliness requirements *composite activities*.

Composite Activities: An Example

Consider an automated toy factory. The factory contains several robots—some of which are equipped for a variety of tasks and some of which are specialized for certain duties. The movement of each robot is directed by a local, low-level control system. These low-level controllers are, in turn, operated by a distributed supervisory real-time system that is responsible for coordinating the robots and for guiding overall production.

Among other things, the factory produces toy fire trucks. The individual components for the trucks (chassis, body, fire ladder, and tires) are fabricated at another site and brought to the automated factory for assembly. The steps in assembling a truck are: 1) attaching the ladder to the body, 2) applying glue to the chassis, 3) joining the body with the chassis, and 4) attaching the wheels. One type of robot is responsible for attaching the ladder and gluing the body and chassis together, while a second robot is specialized for attaching the wheels. Because of its special input/output requirements, the wheel robot is controlled by a separate processing node.

The assembly of each fire truck is controlled by a single high-level activity in the distributed control system. In general (although it may vary with demand), the assembly of a truck should be completed in four minutes. After directing the main assembly robot to pick up the appropriate parts and attach the ladder, the robot is instructed to apply glue to the chassis. It is best to let the glue to dry for 30 seconds before joining the parts. The drying time allows the glue to become "tacky," reducing the chance of a defective bond. Because glue starts drying as soon as it is applied, it is also important to join the body to the chassis within a certain time. Otherwise, the truck may fall apart. While it is best if the parts are joined within 60 seconds, it is acceptable to wait as long as 90 seconds. The penalty for waiting is that more defective trucks may be produced, potentially reducing profits. If more than 90 seconds elapse, the partially completed truck is discarded. Once the truck chassis is assembled, it is passed to the second robot which attaches the wheels.

When one or more of the robots is broken or demand for the trucks is high, the assembly timing requirements are adjusted to reduce the glue drying time and reduce the overall time allowed for completing the assembly.

The truck assembly activity has several significant features. First, it consists of multiple stages (*i.e.*, attaching the ladder, applying the glue, joining the body to the chassis, and attaching the wheels) and involves processing on more than one node. Second, it involves a time-constrained assembly stage (gluing and joining) which has a hard cut-off. This constraint, however, is not a classical hard deadline since an interval of decreasing utility precedes the cut-off time. Third, the time constraint for joining the chassis and body is nested within a higher level timeliness requirement that specifies the total assembly time for the truck. Finally, the system load and timeliness requirements may change because of increased demand or equipment failures.

Because of the potentially complex and dynamic nature of activities and their time constraints, effective processor scheduling for supervisory real-time systems is very difficult. The scheduling problem is further complicated by the requirement that the systems cope gracefully with both transient and permanent overloads caused by changes in the environment (e.g., alarm conditions) or by reductions in the available resources (e.g., node failures). The goals for scheduling resources under these conditions can be summarized as follows:

- When sufficient resources are available, activities should be scheduled in such a way that their timeliness requirements are satisfied.
- When the system is overloaded, activities should be temporarily removed from the schedule so that the timeliness requirements of activities that do execute will be satisfied. Further, the scheduler should choose activities to execute so that the ones selected will be the most beneficial to the application.

The above goals are distinct in several ways from those often suggested for low-level real-time systems. In particular, these goals consider both the timeliness requirements and the relative value of different activities in describing how scheduling decisions should be made. The goals also recognize that activities with soft time constraints may, in some circumstances (e.g., alarm conditions), be more valuable to the application than those with hard deadlines.

1.2 Related Work

Much of the previous real-time scheduling research is based on different assumptions about the application environment and the associated scheduling requirements. Many researchers have considered systems where the workload is very predictable. Other work has concentrated on trying to guarantee hard deadlines under normal conditions—often at the expense of proper overload handling. The research which has investigated more flexible overload handling does not address the composite nature of activities and was not designed for distributed environments.

A significant amount of research has explored the use static priority assignments to meet the real-time requirements of an application. Liu and Layland [Liu 73] described a technique known as *rate monotonic scheduling* in which static priorities are used to schedule periodic tasks that have hard deadlines. In [Sha 86] the technique of *period transformation* is suggested as a method of achieving better overload behavior in these periodic systems. Lehoczky, Sha, and Strosnider ([Lehoczky 87], [Strosnider 88]) have explored how server tasks can be used in a rate monotonic environment to provide fast service for aperiodic tasks that do not have explicit time constraints. This work has been extended in [Sprunt 88] to consid-

er aperiodic tasks with hard deadlines. Rate monotonic scheduling has also been used as the basis for work at the University of Illinois ([Lin 87], [Liu 87], [Chung 88]), where researchers have considered methods for scheduling computations that may yield *imprecise results*.

While advances in priority scheduling have been promising, several factors limit its use in supervisory real-time systems. The static scheduling techniques require a significant amount of *a priori* knowledge about task arrival rates and times. It is often not possible to know this information in dynamic environments. The approach also does not distinguish between the timeliness requirements and the importance of individual tasks. An activity is not necessarily urgent just because it is very important. Nor, is it very important merely because it is urgent. Because both the urgency and importance must be statically encoded into a single value, priority-based schemes are, in general, unable to support the kind of dynamic normal-case scheduling and overload handling that is needed in the supervisory real-time domain.

The second major area of real-time scheduling research has investigated methods of using explicit deadlines to schedule real-time activities. In most cases this work has only considered hard deadlines. The *earliest deadline (ED)* algorithm has been shown to be optimal for uniprocessors by [Dertouzos 74]. Unfortunately, the basic ED scheduling algorithm is unstable under overload conditions [Conway 67]. Several researchers have considered methods of improving the overload behavior of deadline scheduling. At the University of Massachusetts, [Ramamritham 84] has developed dynamic scheduling techniques which will only accept a task for execution if it can be guaranteed to meet its deadline. However, the consequence of this guarantee is that extremely important activities may be blocked by less-important activities that have already been scheduled. To handle this problem, the semantics of the guarantee have been relaxed in [Biyabani 88a] and [Biyabani 88b] to permit higher-priority tasks to revoke guarantees made to lower-priority ones.

Locke developed a scheduling algorithm known as the *best-effort (LBE)* algorithm [Locke 86] which handles overloads in a manner more consistent with the goals of supervisory real-time systems [Jensen 85]. A Mach implementation and complexity evaluation of Locke's original algorithm is described in [Wendorf 88]. An improved best-effort algorithm, implemented for the Alpha operating system [Northcutt 88b], has been evaluated in [Trull 88]. In related work, Clark has developed an approach to scheduling dependent real-time activities with similar goals [Clark 88]. Although previous work in this area has been very successful, limitations still exist. In particular, previous work has not adequately considered how physical distribution or nested timeliness requirements affect the scheduling problem.

Little research has been conducted to investigate the effects of physical distribution on the scheduling of composite real-time activities. Most distributed scheduling papers describe load-sharing techniques, but never consider the possibility that a time-constrained activity may span multiple nodes. Examples of results which fall into this category are [Ramamritham 84], [Stankovic 84], [Stankovic 85], [Zhao 85], [Kurose 86], and [Kurose 87]. The most closely related research addresses the scheduling of groups of precedence-related tasks with hard deadlines [Cheng 86]. Cheng describes methods of synthesizing intermediate time constraints known as *pseudo windows* to ensure that tasks in a group are scheduled to satisfy the group's deadline.¹ In general, the pseudo window technique requires that complete knowledge about the execution characteristics of all group members be available when the first member of the group arrives. Although such extensive knowledge does allow more intelligent scheduling decisions to be made, the system dynamics often limit the extent to which this information is available.

Related scheduling work in the field of operations research has addressed the problem of multistage production planning [Johnson 74]. Unfortunately, techniques such as linear programming [Winston 87] which are often employed in these situations are not practical for on-line scheduling.

1.3 Technical Approach

As the previous section indicates, existing research has addressed only parts of the supervisory real-time scheduling problem. This work will extend the range of that coverage to include the scheduling of composite real-time activities—that is, time-constrained activities that may span multiple processing nodes and may have multiple nested timeliness requirements.

This research will be divided into three major stages:

- creation of a computational model for composite real-time activities,
- development of time-driven scheduling techniques for composite activities, and
- analysis and evaluation of the proposed techniques.

The following sections describe in greater detail the work involved and results to date in each of these areas

¹ This use of pseudo windows should not be confused with the case where nested time constraints are actually imposed by the application.

2. Computational Model

The first stage of the research involves the development of a new computational model which can be used to describe the behavior and timeliness requirements of composite real-time activities. The model must account for the physical distribution of time-constrained activities and must specify meaningful methods for composing multiple nested timeliness requirements.

2.1 Modeling Timeliness Requirements

This work uses the notion of *time-value functions* [Jensen 75] to specify the timeliness requirements of activities. Time-value functions express the time-varying value to the application of completing an activity. Some example specifications are shown in Figure 1.. Using this model, a hard deadline (Figure 1a) is specified by a step function where the completion value is a constant positive number between the *request time* (t_r) and the deadline (t_{dl}), and is zero after the deadline. The glue/join activity described in the toy factory example would have a time value function similar to Figure 1b, where the utility of finishing the work increas-

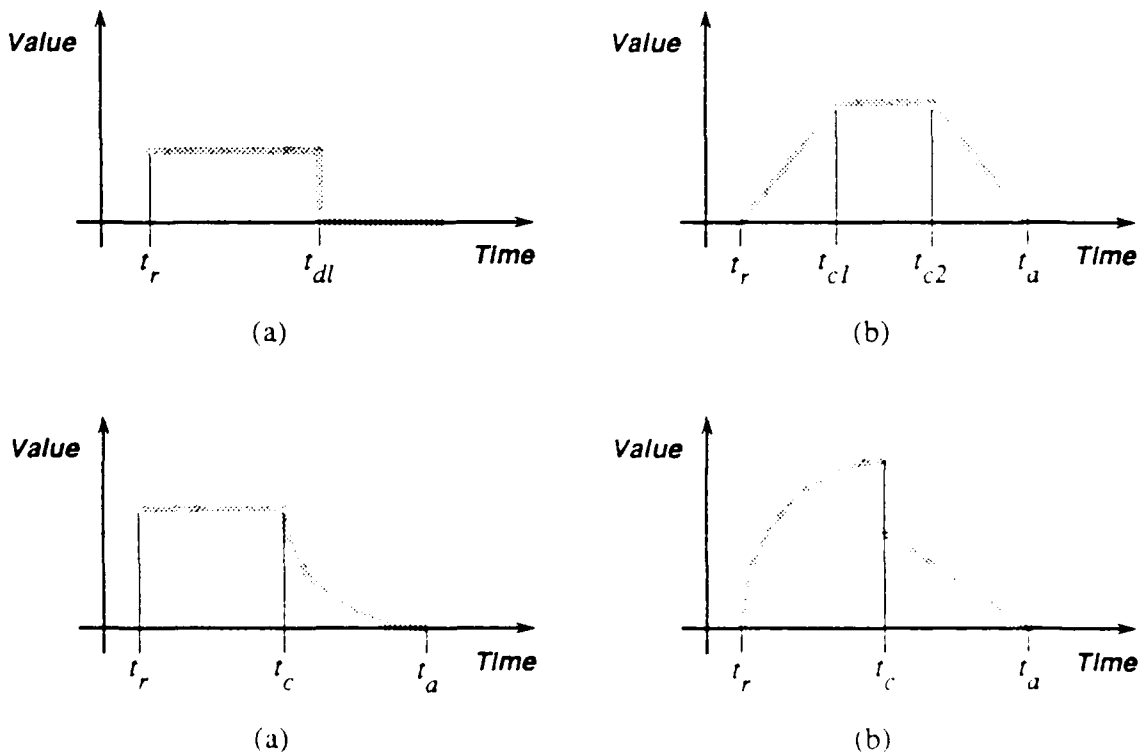


Figure 1: Example Time-Value Functions

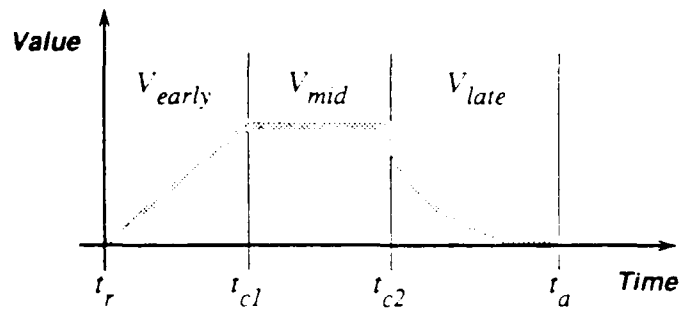


Figure 2: Time-Value Function Specification

es (as the glue becomes tacky) until a *critical time* (t_{c1}), is constant for an interval (until t_{c2} , when the glue begins to dry), and gradually decreases to zero afterward (when the partially assembled truck must be discarded at t_a , the *abort time*).

In theory, time-value functions may have arbitrary shapes. To simplify their specification in real systems such as [Locke 86], [Tokuda 87], and [Northcutt 88b], time-value functions are often modeled by a set of continuous functions and reference times. In this work, time-value functions will be specified by three reference times (t_r , t_{c1} , and t_{c2}), and three continuous functions (V_{early} , V_{mid} , and V_{late}) (see Figure 2). In the most general case, each of the functions has the form:

$$V(t) = K_1 + K_2t + K_3t^2 + K_4e^{-K_5t}$$

In most cases, however, this research will consider only linear functions (*i.e.*, $K_3=K_4=0$). This restriction still allows most interesting time-value functions to be approximated, yet greatly simplifies the mathematics which must be handled by the scheduler. The abort time (t_a), the time after which the activity is aborted and any exception processing is initiated, is defined as the earliest time when $V_{late} = 0$.

When timeliness requirements are nested, appropriate techniques of composing the time-value functions must be developed. It is not sufficient to consider only the innermost requirement since that constraint may, in fact, not be the most stringent. Nor may it be appropriate to use only the most stringent constraint when deciding on the ultimate value of an activity. In the toy truck example, there may be no inherent value in successfully gluing the chassis and body together if the wheels are never attached. At present, work is continuing on efforts to identify appropriate methods of composing nested time constraints.

2.2 Modeling Composite Activities

Previous time-driven scheduling research has only considered activities that remain locally executable from the time they arrive until they complete. Under these conditions, the execution characteristics of an activity can be modeled by a single statistic, the *estimated computation time (ECT)*. As explained in the truck assembly problem, composite real-time activities may involve computation on several processing nodes. Because of this distribution, the total computation time will also be divided among multiple nodes. To completely describe the execution characteristics of a distributed activity, the computation time on each node must be specified.

One useful way of modeling the distributed activities is to view them as having multiple stages—each of which may execute on a different processing node. An activity can then be described as a linear-connected graph of execution stages. Time constraint specifications can be modeled by adding graph elements corresponding to the beginning and end of each time constraint. Figure 3 illustrates how the toy truck assembly activity is modeled using this technique. For each execution stage, the estimated computation time, $ECT(n)$, of that stage is specified.

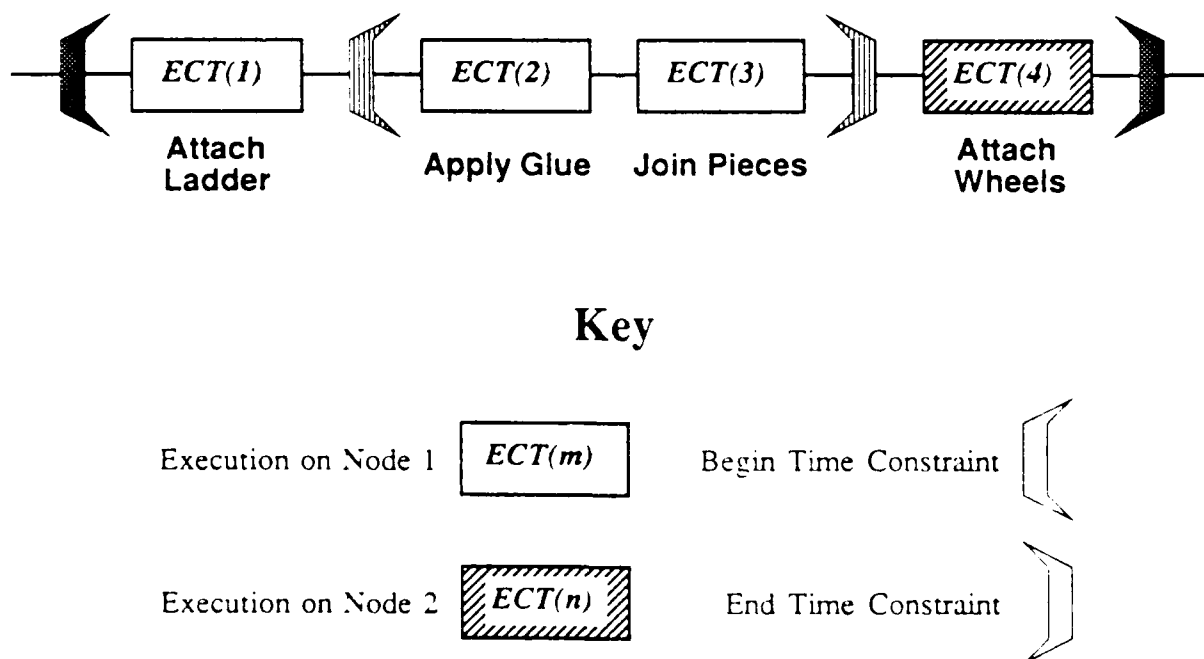


Figure 3: Graph Model of Toy Truck Assembly

Although the full generality of the graph-based model is not strictly needed to handle distributed activities, the model is useful since it can be easily extended to include general resource requirements and the concurrent execution of component stages of an activity.

3. Scheduling Techniques

Once the computational model has been finalized, new time-driven scheduling techniques will be developed. These techniques will use limited information about the timeliness requirements and execution characteristics of composite real-time activities to ensure that as much useful work as possible is completed by the application. Under normal (*i.e.*, non-overload) conditions, the algorithms will attempt to satisfy all of the system timeliness requirements. When overloads occur, the algorithms will be designed to shed load intelligently so that activities which are scheduled will meet their time constraints and will contribute as much value as possible to the system.

The scheduling problem can be divided into three major components:

- ordering of activities,
- overload detection, and
- load shedding.

Several techniques have been suggested for ordering time-constrained activities. Both ED and *least slack (LS)* ordering are known to be optimal for uniprocessors [Mok 83]. Unfortunately, neither approach is optimal in the multiprocessor case. Simulation results [Locke 86] have shown, however, that ED scheduling still performs well on multiprocessors. [Stone 77] suggests that network flow algorithms can be used effectively for processor scheduling. Still another approach handles task ordering as a planning problem where search techniques are used to find an acceptable execution order [Ramamritham 84]. These search techniques are imperfect, but tractable. Several task ordering alternatives will be considered for possible inclusion in the new scheduling algorithms. The most likely candidates include ED ordering, and an explicit placement algorithm which allows varying levels of "greediness" to be applied.

Overload detection relies on the use of execution time estimates to determine whether there are enough processor cycles to schedule all contending activities within their time constraints. In many dynamic real-time systems, this calculation is performed during the task ordering process by calculating the cumulative slack time for the processor. If the cumulative slack for any time interval drops below zero, the processor is overloaded. The problem of overload detection is complicated by distributed computations. If the distribution is not considered, the local demand for the processor will be overestimated. This exaggeration could cause "false" overload indications to be generated, potentially preventing activities from meeting their time constraints (due to load shedding).

The problem of false overloads can be ameliorated by incorporating additional knowledge into the overload detection process. The alternative which uses the least additional execution information is one in which estimates for both *local computation time (LCT)* and *total computation time (TCT)* are specified as part of a time constraint. In the general case, the activity must specify LCT's for each of the nodes it may visit. The local scheduler would continue to use the TCT figure to determine the ordering of the schedule, but would use the LCT figure when performing slack time calculations for the processor. The use of LCT and TCT estimates should reduce the number of false overload signals. However, false indications may still occur since LCT/TCT estimates do not specify when in an activity's execution the processor time will be needed. If most of the required time comes late in the activity, then more near-term (and fewer far-term) processor cycles may be available than the LCT/TCT ratio would indicate. More accurate overload detection could be achieved if ECT's were available for each stage of an activity's execution. However, additional processing would be required to process the additional information. It might also be impractical or impossible to gather such detailed execution profiles.

Load shedding in time-driven systems relies on the notion of *value density* [Locke 86] to determine which activities are likely to contribute the most to the application. Value density measures how much value per unit of processing time will be returned to the application for executing a particular activity. When a limited number of processor cycles are available, executing the activities with the highest value density is likely to result in the most useful work being accomplished. In a distributed environment, load shedding decisions are more difficult since an activity with a relatively low value density might only require a few cycles locally before travelling to a more lightly-loaded node where its time constraints could be satisfied. The load shedding algorithms should be able to utilize the same types of extended execution information (LCT, TCT) as the overload detection algorithms, although a better solutions would like require cooperation between schedulers at different nodes.

The existence of nested timeliness requirements complicates load shedding by making it more difficult to determine the value density. Since the value of satisfying a time constraint may depend both on the innermost constraint and on outer-level constraints, a composition function must be applied to determine the true potential value for the activity. Work is in progress which will specify methods of composing time-value functions.

Research into the specific scheduling algorithms is still in its initial stages. Many of the tough decisions will be easier to make once the computational model has been finalized and techniques for composing nested time-value functions have been developed.

4. Evaluation Methodology

The evaluation of the proposed scheduling techniques will judge how well they satisfy the requirements of the supervisory real-time environment. Simulation experiments will be used to evaluate the performance of the new algorithms compared to existing methods such as earliest deadline and Locke's best-effort approach.

Because the execution of time-driven scheduling algorithms can be costly compared to other algorithms, scheduling overhead will be explicitly considered in the simulation experiments. There are two primary techniques for estimating scheduling overhead: 1) mathematically deriving the theoretical complexity of the scheduling algorithm and 2) measuring the actual performance of a sample implementation. Both techniques yield useful results and will be employed in this research. The complexity analysis provides worst-case information describing how the overhead changes with increasing load. Simple $O(n)$ analysis, however, does not reveal the magnitude of the constant and scale factors. Actually measuring the performance of a sample implementation provides valuable information about these scale factors (compared with other algorithms) and reveals how the algorithm behaves on common (*i.e.*, non-worst-case) workloads.

Evaluating the proposed techniques using only one or two "real" workloads would likely skew the results to reflect the peculiarities of the chosen systems. For this reason, a wide range of synthetic workloads will be used to test the performance of the algorithms. Sensitivity analysis will be used to measure how their behavior changes as different components of the workload are varied.

4.1 Evaluation Metrics

Many metrics have been devised for evaluating the performance of real-time scheduling algorithms. For time-driven systems, the most important measure of an algorithm's performance is how much of an application's potential value is obtained using the algorithm. Since the fraction of the maximum completion value obtained for each activity directly corresponds to how well its time constraints have been satisfied, the value metric indicates both how well the system time constraints have been satisfied, and to what degree the scheduler has succeeded in scheduling the most useful activities under overload conditions. Because optimal scheduling is intractable in the systems of interest, it is not practical to compute the maximum value that could actually be garnered by a perfect scheduling algorithm. Instead, a simple upper bound on the potential value can be generated by summing the maxima of the val-

ues for the activities in a workload. This *total potential value (TPV)* is then compared to the *actual value obtained (AVO)* to yield the *percentage of value obtained (PVO)* by the algorithm.

Several other metrics have been designed to measure how well an algorithm satisfies the goals of lower-level systems and are not directly applicable to the supervisory real-time environment. However, some of these low-level metrics can be adjusted to provide useful information about algorithm behavior for certain workloads.

In systems where all tasks are presumed to have deadlines and are not distinguished by value, the performance of scheduling algorithms is often judged by monitoring the percentage of tasks which complete after their deadlines (*i.e.*, are *tardy*). In these cases, the *mean tardiness* and/or the *maximum tardiness* of the late tasks are often considered as well. For time-driven systems, the *percentage of tardy activities (PTA)* is only relevant if it is theoretically possible to schedule the workload being considered so that it obtains its total potential value. Under these conditions, the PTA metric can provide useful information about the behavior of the algorithm. Statistics on mean tardiness and maximum tardiness do not consider the relative importance of the late activities and, therefore, are not relevant.

Other metrics which may provide useful information for evaluating the behavior of the proposed algorithms include:

- *average scheduling overhead*—the average time required to choose the next activity to run, and
- *number of preemptions*—the total number of preemptions generated (useful for estimating context swap overhead).

4.2 Workload Generation

A synthetic workload generator will be used to construct test cases for the simulation experiments. By using synthetic workloads, a wide range of application environments can be simulated. [Woodbury 86] has investigated methods of developing workloads for distributed real-time systems. Although the computational model considered by Woodbury is more constrained than the one presented in Section 2, the work does explore the issues which lead to stochastic behavior of distributed activities. Other work such as [Maynard 88] will be used to identify important characteristics of supervisory real-time applications which should be modeled in the workloads.

Simulation experiments will be conducted while varying such factors as:

- mean number of activities,
- mean activity arrival rates,

- mean and standard deviation of activity execution times,
- percentage of local versus remote execution times, and
- level of system "dynamics" (*i.e.*, how the characteristics of the workload change across time).

Sensitivity analysis will be used to determine which workload variations have the greatest impact on the behavior of the various algorithms. Using these results, it should be possible to characterize the types of workloads which are best handled by the proposed techniques. These results will also be used to suggest techniques for tailoring the proposed algorithms to more closely match certain environments.

5. Research Schedule

- 8/30/89 — Thesis proposal.
- 10/31/89 — A computational model will have been developed and shown to be suitable for modeling composite real-time activities and their timeliness requirements. Techniques for using the information available from the expanded model will have been proposed.

The specification of the composite activity execution model is largely complete. More work is needed to formalize techniques for composing nested time constraints and to identify what types of processor utilization information can be made available to the scheduler at run time.

- 12/31/89 — Specific scheduling algorithms will have been developed. A complexity analysis of the proposed algorithms will have also been completed.

Initial investigations have been made to identify possible scheduling techniques. Although tentative, this work indicates that promising techniques do exist.

- 2/28/90 — A simulator and synthetic workload generator will have been designed and implemented.
- 5/31/90 — Simulation experiments will have been completed. The proposed scheduling algorithms will have been evaluated and refined to the point where final analyses can be performed.
- 8/31/90 — Data from the simulation work will have been analyzed and a thesis describing the results of the research will have been completed.

6. Research Contribution

The contributions of this research can be classified into three major categories—the creation of a computational model for composite real-time activities, the development of scheduling techniques suitable for the supervisory real-time environment, and the analysis and evaluation of scheduling techniques that are suitable for different types of supervisory systems.

A new computational model will be developed for describing composite real-time activities. This model will account for the physical distribution of time-constrained activities and will specify suitable methods for composing multiple nested timeliness requirements. No existing models can adequately describe the behavior and requirements of such activities. The computational model will be designed so that it can be easily extended to consider both general resource requirements and the concurrent execution of component stages.

New time-driven scheduling techniques will be developed. These techniques will rely on limited information about the timeliness requirements and execution characteristics of composite real-time activities to schedule activities so that as much useful work as possible is completed by the application. Under normal (*i.e.*, non-overload) conditions, the algorithms will attempt to satisfy all of the system timeliness requirements. When overloads occur, the algorithms will be designed to shed load intelligently so that activities which are scheduled will meet their time constraints and will contribute as much value as possible to the system.

Finally, the proposed scheduling methods will be analyzed under a wide range of loading conditions. The performance of the new algorithms will be compared to that obtained using existing approaches. The results of this analysis will be used to delineate the domain in which the new techniques are applicable and to suggest methods for tailoring the algorithms to specific types of real-time environments.

Bibliography

- [Alger 86] L. S. Alger and J. H. Lala.
A Real-Time Operating System for a Nuclear Power Plant Computer.
In *Proceedings Real-Time Systems Symposium*, pages 244-248. IEEE Computer Society Press, December, 1986.
- [Belzile 86] C. Belzile, M. Coulas, G. H. MacEwen, and G. Marquis.
RNet: A Hard Real-Time Distributed Programming System.
In *Proceedings Real-Time Systems Symposium*, pages 2-13. IEEE Computer Society Press, December, 1986.
- [Biyabani 88a] S. R. Biyabani, J. A. Stankovic, and K. Ramamritham.
The Integration of Deadline and Criticalness in Hard Real-Time Scheduling.
In *Proceedings Real-Time Systems Symposium*, pages 152-160. IEEE Computer Society Press, December, 1988.
- [Biyabani 88b] S. Biyabani, J. A. Stankovic, and K. Ramamritham.
The Integration of Deadline and Criticalness Requirements in Hard Real-Time Systems.
In *Abstracts of IEEE and USENIX of the Fifth Workshop on Real-Time Software and Operating Systems*, pages 12-17. IEEE Computer Society, May, 1988.
- [Bond 88] R. Bond, S. Bemrich, J. Connelly, G. Pendergrass, J. Hulsey.
Missile Guidance Processor Software Development A Case Study.
In *Proceedings Real-Time Systems Symposium*, pages 60-68. IEEE Computer Society Press, December, 1988.
- [Bourne 84] D. A. Bourne and M. S. Fox.
Autonomous Manufacturing: Automating the Job-Shop.
Computer :76-86, September, 1984.
- [Chang 85] H.-Y. Chang and M. Livny.
Priority in Distributed Systems.
In *Proceedings Real-Time Systems Symposium*, pages 123-130. IEEE Computer Society Press, December, 1985.
- [Chang 86] H.-Y. Chang and M. Livny.
Distributed Scheduling under Deadline Constraints: a Comparison of Sender-initiated and Receiver-initiated Approaches.
In *Proceedings Real-Time Systems Symposium*, pages 175-180. IEEE Computer Society Press, December, 1986.
- [Cheng 86] S. Cheng, J. A. Stankovic, and K. Ramamritham.
Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems.
In *Proceedings Real-Time Systems Symposium*, pages 166-174. IEEE Computer Society Press, December, 1986.

- [Cheng 87] S.-C. Cheng, J. A. Stankovic, and K. Ramamritham.
Scheduling Algorithms for Hard Real-Time Systems -- A Brief Survey.
Real-Time Systems Newsletter 3(2):1-24, Summer, 1987.
- [Chu 84] W. W. Chu and K. K. Leung.
Task Response Time Model & Its Applications for Real-Time Distributed
Processing Systems.
In *Proceedings Real-Time Systems Symposium*, pages 225-236. IEEE
Computer Society Press, December, 1984.
- [Chu 88] W. W. Chu and C. M. Sit.
Estimating Task Response Time with Contentions for Real-Time Distributed
Systems.
In *Proceedings Real-Time Systems Symposium*, pages 272-281. IEEE
Computer Society Press, December, 1988.
- [Chung 88] J.-Y. Chung and J. W. S. Liu.
Algorithms for Scheduling Periodic Jobs to Minimize Average Error.
In *Proceedings Real-Time Systems Symposium*, pages 142-151. IEEE
Computer Society Press, December, 1988.
- [Clark 88] P. K. Clark.
Scheduling Dependent Real-Time Activities.
Ph.D. Proposal, School of Computer Science, Carnegie Mellon University.
October, 1988
- [Conway 67] R. W. Conway, W. L. Maxwell, and L. W. Miller.
Theory of Scheduling.
Addison-Wesley, 1967.
- [Coulas 87] M. F. Coulas, G. H. Macewen, and G. Marquis.
RNet: A Hard Real-Time Distributed Programming System.
IEEE Transactions on Computers C-36(8):917-932, August, 1987.
- [Daniels 86] D. C. Daniels and H. F. Wedde.
Real-Time Performance of a Completely Distributed Operating System.
In *Proceedings Real-Time Systems Symposium*, pages 157-163. IEEE
Computer Society Press, December, 1986.
- [Davari 86] S. Davari and S. K. Dhall.
An On Line Algorithm for Real-Time Tasks Allocation.
In *Proceedings Real-Time Systems Symposium*, pages 194-200. IEEE
Computer Society Press, December, 1986.
- [Davidson 89] S. Davidson, I. Lee, and V. Wolfe.
A Protocol for Timed Atomic Commitment.
In *Proceedings of the 9th International Conference on Distributed Computing
Systems*, pages 199-206. IEEE Computer Society Press, June, 1989.
- [Dertouzos 74] M. Dertouzos.
Control Robotics: The Procedural Control of Physical Processes.
In *Proceedings of the IFIP Congress*, pages 807-813. IFIP, 1974.

- [Donner 86] M. D. Donner and D. H. Jameson.
A Real-Time Juggling Robot.
In *Proceedings Real-Time Systems Symposium*, pages 249-256. IEEE Computer Society Press, December, 1986.
- [Efe 89] K. Efe and B. Groselj.
Minimizing Control Overheads in Adaptive Load Sharing.
In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 307-315. IEEE Computer Society Press, June, 1989.
- [Fleisch 84] B. D. Fleisch.
Meta-Activities: Towards Coherent Distributed Jobs.
In *Proceedings of the 4th International Conference on Distributed Computing Systems*, pages 566-578. IEEE Computer Society Press, May, 1984.
- [Fortier 85] P. J. Fortier.
Design and Analysis of Distributed Real-Time Systems.
Intertext Publications, McGraw-Hill, 1985.
- [Gabrielian 88] A. Gabrielian and M. K. Franklin.
State-Based Specification of Complex Real-Time Systems.
In *Proceedings Real-Time Systems Symposium*, pages 2-11. IEEE Computer Society Press, December, 1988.
- [Hong 88] K. S. Hong and J. Y-T. Leung.
On-Line Scheduling of Real-Time Tasks.
In *Proceedings Real-Time Systems Symposium*, pages 244-250. IEEE Computer Society Press, December, 1988.
- [Jahanian 87] F. Jahanian and A. K.-L. Mok.
A Graph-Theoretic Approach for Timing Analysis and its Implementation.
IEEE Transactions on Computers C-36(8):961-975, August, 1987.
- [Jensen 75] E. D. Jensen.
Time-Value Functions for BMD Radar Scheduling.
Technical Report, Honeywell Systems and Research Center, June, 1975.
- [Jensen 85] E. D. Jensen, C. D. Locke, and H. Tokuda.
A Time-Driven Scheduling Model for Real-Time Operating Systems.
In *Proceedings Real-Time Systems Symposium*, pages 112-122. IEEE Computer Society Press, December, 1985.
- [Johnson 77] L. A. Johnson and D. C. Montgomery.
Operations Research in Production Planning, Scheduling, and Inventory Control.
John Wiley & Sons, 1977.
- [Kligerman 86] E. Kligerman and A. D. Stoyenko.
Real-Time Euclid: A Language for Reliable Real-Time Systems.
IEEE Transactions on Software Engineering SE-12(9):941-949, September, 1986.

- [Krueger 87] P. Krueger and M. Livny.
The Diverse Objectives of Distributed Scheduling Policies.
In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 242-249. IEEE Computer Society Press, September, 1987.
- [Kurose 86] J. F. Kurose, S. Singh, and R. Chipalkatti.
A Study of Quasi-Dynamic Load Sharing in Soft Real-Time Distributed Computer Systems.
In *Proceedings Real-Time Systems Symposium*, pages 201-208. IEEE Computer Society Press, December, 1986.
- [Kurose 87] J. F. Kurose and R. Chipalkatti.
Load Sharing in Soft Real-Time Distributed Computer Systems.
IEEE Transactions on Computers C-36(8):993-1000, August, 1987.
- [Lee 85] I. Lee and V. Gehlot.
Language Constructs for Distributed Real-Time Programming.
In *Proceedings Real-Time Systems Symposium*, pages 57-66. IEEE Computer Society Press, December, 1985.
- [Lee 87] I. Lee and S. B. Davidson.
Adding Time to Synchronous Process Communications.
IEEE Transactions on Computers C-36(8):941-948, August, 1987.
- [Lehoczky 87] J. P. Lehoczky, L. Sha, and J. K. Strosnider.
Enhanced Aperiodic Responsiveness in Hard Real-Time Environments.
In *Proceedings Real-Time Systems Symposium*, pages 261-270. IEEE Computer Society Press, December, 1987.
- [Leinbaugh 86] D. W. Leinbaugh and M.-R. Yamini.
Guaranteed Response Times in a Distributed Hard-Real-Time Environment.
IEEE Transactions on Software Engineering SE-12(12):1139-1144, December, 1986.
- [Lin 87] K.-J. Lin, S. Natarajan, and J. W.-S. Liu.
Imprecise Results: Utilizing Partial Computations in Real-Time Systems.
In *Proceedings Real-Time Systems Symposium*, pages 210-217. IEEE Computer Society Press, December, 1987.
- [Lin 88] K.-J. Lin.
Expressing and Maintaining Timing Constraints in FLEX.
In *Proceedings Real-Time Systems Symposium*, pages 96-105. IEEE Computer Society Press, December, 1988.
- [Liu 73] C. L. Liu and J. W. Layland.
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.
Journal of the ACM 20(1):46-61, January, 1973.
- [Liu 87] J. W. S. Liu, K.-J. Lin, S. Natarajan.
Scheduling Real-Time, Periodic Jobs Using Imprecise Results.
In *Proceedings Real-Time Systems Symposium*, pages 252-260. IEEE Computer Society Press, December, 1987.

- [Lo 87] S. P. Lo and V. D. Gligor.
A Comparative Analysis of Multiprocessor Scheduling Algorithms.
In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 356-363. IEEE Computer Society Press, September, 1987.
- [Locke 86] C. D. Locke.
Best-Effort Decision Making for Real-Time Scheduling.
PhD thesis, Carnegie Mellon University, May, 1986.
- [Maynard 88] D. P. Maynard, R. K. Clark, J. D. Northcutt, S. E. Shipman, R. B. Kegley, P. J. Keleher, B. A. Zimmerman, and E. D. Jensen.
The Alpha Operating System: An Example Command, Control, and Battle Management Application.
Technical Report, Archons Project, School of Computer Science, Carnegie Mellon University, 1988.
- [McNaughton 59] R. McNaughton.
Scheduling with Deadlines and Loss Functions.
Management Science 6(1):1-12, October, 1959.
- [Mirchandaney 89] R. Mirchandaney, D. Towsley, and J. A. Stankovic.
Adaptive Load Sharing in Heterogeneous Systems.
In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 298-306. IEEE Computer Society Press, June, 1989.
- [Mok 83] A. K.-L. Mok.
Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.
PhD thesis, Massachusetts Institute of Technology, May, 1983.
- [Mok 84a] A. K. Mok.
The Design of Real-Time Programming Systems Based on Process Models.
In *Proceedings Real-Time Systems Symposium*, pages 5-17. IEEE Computer Society Press, December, 1984.
- [Mok 84b] A. K. Mok.
The Decomposition of Real-Time System Requirements into Process Models.
In *Proceedings Real-Time Systems Symposium*, pages 125-134. IEEE Computer Society Press, December, 1984.
- [Northcutt 88a] J. D. Northcutt and R. K. Clark.
The Alpha Operating System: Programming Model.
Archons Project 88021, School of Computer Science, Carnegie Mellon University, February, 1988.
- [Northcutt 88b] J. D. Northcutt and R. K. Clark.
The Alpha Operating System: Kernel Internals.
Archons Project 88051, School of Computer Science, Carnegie Mellon University, May, 1988.

- [Olson 86] R. Olson.
Realtime Response on a Message Based Multiprocessor.
In *Proceedings Real-Time Systems Symposium*, pages 28-35. IEEE Computer Society Press, December, 1986.
- [Ostroff 87] J. S. Ostroff and W. M. Wonham.
Modelling, Specifying, and Verifying Real-time Embedded Computer Systems.
In *Proceedings Real-Time Systems Symposium*, pages 124-132. IEEE Computer Society Press, December, 1987.
- [Ostroff 89] J. S. Ostroff.
Verifying finite state real-time discrete event processes.
In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 207-216. IEEE Computer Society Press, June, 1989.
- [Peng 89] D.-T. Peng and K. G. Shin.
Static Allocation of Periodic Tasks with Precedence Constraints in Distributed Real-Time Systems.
In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 190-198. IEEE Computer Society Press, June, 1989.
- [Rajkumar 88] R. Rajkumar, L. Sha, and J. P. Lehoczky.
Real-Time Synchronization Protocols for Multiprocessors.
In *Proceedings Real-Time Systems Symposium*, pages 259-269. IEEE Computer Society Press, December, 1988.
- [Ramamritham 84] K. Ramamritham, J. A. Stankovic.
Dynamic Task Scheduling in Distributed Hard Real-Time Systems.
In *Proceedings of the 4th International Conference on Distributed Computing Systems*, pages 96-107. IEEE Computer Society Press, May, 1984.
- [Ramamritham 87] K. Ramamritham, J. A. Stankovic, and W. Zhao.
Meta-Level Control in Distributed Real-Time Systems.
In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 10-17. IEEE Computer Society Press, September, 1987.
- [Ruschitzka 77] M. Ruschitzka and R. S. Fabry.
A Unifying Approach to Scheduling.
Communications of the ACM 20(7):469-477, July, 1977.
- [Sahni 79] S. Sahni and Y. Cho.
Nearly On Line Scheduling of a Uniform Processor System with Release Times.
SIAM Journal on Computing 8(2):275-285, May, 1979.
- [Saponas 86] T. G. Saponas.
A Real-Time Distributed Processing System.
In *Proceedings Real-Time Systems Symposium*, pages 36-43. IEEE Computer Society Press, December, 1986.

- [Schwan 86] K. Schwan, W. Bo, and P. Gopinath.
A High-Performance, Object-Based Operating System for Real-Time, Robotics Applications.
In *Proceedings Real-Time Systems Symposium*, pages 147-156. IEEE Computer Society Press, December, 1986.
- [Schwan 87] K. Schwan, P. Gopinath, and W. Bo.
CHAOS - Kernel Support for Objects in the Real-Time Domain.
IEEE Transactions on Computers C-36(8):904-916, August, 1987.
- [Sha 83] L. Sha, E. D. Jensen, R. F. Rashid, and J. D. Northcutt.
Distributed Co-operating Processes and Transactions.
In Y. Parker and J.-P. Verjus (editor), *Distributed Computing Systems Synchronization, Control, and Communication*, pages 23-50. Academic Press, 1983.
- [Sha 86] L. Sha, J. P. Lehoczky, and R. Rajkumar.
Solutions for Some Practical Problems in Prioritized Preemptive Scheduling.
In *Proceedings Real-Time Systems Symposium*, pages 181-191. IEEE Computer Society Press, December, 1986.
- [Sprunt 88a] B. Sprunt, J. Lehoczky, and L. Sha.
Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm.
In *Proceedings Real-Time Systems Symposium*, pages 251-258. IEEE Computer Society Press, December, 1988.
- [Sprunt 88b] B. Sprunt.
Aperiodic Task Scheduling for Hard Real-Time Systems.
Ph.D. Proposal, Department of Electrical and Computer Engineering, Carnegie Mellon University.
November, 1988
- [Stankovic 84] J. A. Stankovic and I. S. Sidhu.
An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups.
In *Proceedings of the 4th International Conference on Distributed Computing Systems*, pages 49-59. IEEE Computer Society Press, May, 1984.
- [Stankovic 85] J. A. Stankovic, K. Ramamritham, and S. Cheng.
Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems.
IEEE Transactions on Computers C-34(12):1130-1143, December, 1985.
- [Stankovic 87] J. A. Stankovic and K. Ramamritham.
The Design of the Spring Kernel.
In *Proceedings Real-Time Systems Symposium*, pages 146-157. IEEE Computer Society Press, December, 1987.
- [Stone 77] H. S. Stone.
Multiprocessor Scheduling with the Aid of Network Flow Algorithms.
IEEE Transactions on Software Engineering SE-3(1):85-93, January, 1977.

- [Strosnider 88] J. K. Strosnider.
Highly Responsive Real-Time Token Rings.
PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, August, 1988.
- [Thurber 73] K. J. Thurber and L. A. Jack.
Time-Driven Scheduling.
In *Digest of Papers COMPCON73*, pages 181-184. IEEE Computer Society Press, February, 1973.
- [Tokuda 87] H. Tokuda, J. W. Wendorf, H.-Y. Wang.
Implementation of a Time-Driven Scheduler for Real-Time Operating Systems.
In *Proceedings Real-Time Systems Symposium*, pages 271-280. IEEE Computer Society Press, December, 1987.
- [Tokuda 88] H. Tokuda and M. Kotera.
A Real-Time Tool Set for the ARTS Kernel.
In *Proceedings Real-Time Systems Symposium*, pages 289-299. IEEE Computer Society Press, December, 1988.
- [Trull 88] J. E. Trull, J. D. Northcutt, R. K. Clark, S. E. Shipman, D. P. Maynard, and D. C. Lindsay.
An Evaluation of the Alpha Real-Time Scheduling Policies.
Archons Project 88102, School of Computer Science, Carnegie Mellon University, October, 1988.
- [Wendorf 88] J. W. Wendorf.
Implementation and Evaluation of a Time-Driven Scheduling Processor.
In *Proceedings Real-Time Systems Symposium*, pages 172-180. IEEE Computer Society Press, December, 1988.
- [Winston 87] W. L. Winston.
Operations Research: Applications and Algorithms.
PWS Publishers, 1987.
- [Woodbury 86] M. H. Woodbury.
Analysis of the Execution Time of Real-Time Tasks.
In *Proceedings Real-Time Systems Symposium*, pages 89-96. IEEE Computer Society Press, December, 1986.
- [Zhao 85] W. Zhao and K. Ramamritham.
Distributed Scheduling using Bidding and Focused Addressing.
In *Proceedings Real-Time Systems Symposium*, pages 103-111. IEEE Computer Society Press, December, 1985.
- [Zhao 87] W. Zhao, K. Ramamritham, and J. A. Stankovic.
Preemptive Scheduling Under Time and Resource Constraints.
IEEE Transactions on Computers C-36(8):949-960, August, 1987.

Scheduling Dependent Real-Time Activities

Raymond K. Clark

*School of Computer Science
Carnegie Mellon University*

February 6, 1990

Table of Contents

1. Introduction	C-1
1.1. Problem Definition	C-2
1.1.1. Dependencies	C-3
1.1.2. Real-Time Systems	C-4
1.2. Simple and Complex Schedulers	C-6
1.3. Scheduling Example	C-9
1.4. Motivation for the Model	C-10
1.4.1. Accrued Value	C-10
1.4.2. Time-Value Functions	C-13
1.5. Technical Approach	C-14
1.5.1. Define Model	C-14
1.5.2. Devise Algorithms	C-14
1.5.3. Prove Properties Analytically	C-15
1.5.4. Simulate Algorithm	C-15
2. The Scheduling Model	C-17
2.1. Informal Model and Rationale	C-17
2.1.1. Applications, Activities, and Phases	C-17
2.1.2. Shared Resources	C-18
2.1.3. Phase Preemption	C-18
2.1.4. Phase Abortion	C-19
2.1.5. Events	C-19
2.1.6. Histories	C-20
2.1.7. Scheduling Automata	C-21
2.1.7.1. General Structure	C-21
2.1.7.2. Specific Scheduling Automata	C-23
2.2. Assumptions and Restrictions of Model	C-24
2.3. Formal Model	C-24
2.3.1. Notation and Definitions	C-24
2.3.2. The General Scheduling Automaton Framework (GSAF)	C-26
2.3.2.1. Applications and Activities	C-26
2.3.2.2. Events and Histories	C-26
2.3.2.3. Operations	C-26
2.3.2.4. Computational Phases of Activities	C-28
2.3.2.5. Shared Resources	C-28
2.3.2.6. Phase Preemption and Resumption	C-28
2.3.2.7. Event Terminology and Notation	C-28
2.3.2.8. Definitions and Properties of Histories	C-29
2.3.2.9. Automaton State Components	C-30
2.3.2.10. Operations Accepted by GSAF with Preconditions and Postconditions	C-33
2.3.2.11. Active Phase Selection	C-36
2.3.3. Notes	C-37
2.3.3.1. Manifestation of Assumptions and Restrictions	C-37
2.3.3.2. Manifestation of Interrupts	C-38

2.3.3.3. Atomic Nature of 'Request-Phase' Events	C-38
2.4. Observations on the Model	C-38
3. The DASA Algorithm	C-41
3.1. Dependent Activity Scheduling Algorithm	C-41
3.1.1. Rationale for Heuristics	C-41
3.1.2. The DASA Algorithm	C-42
3.1.3. The DASA Algorithm: Dependency Scheduling	C-42
3.1.4. The DASA Algorithm: Deadlock Resolution	C-44
3.2. Formal Definition of DASA	C-44
3.2.1. The Formal Definition	C-44
3.2.1.1. DASA Automaton State Components	C-44
3.2.1.2. Operations Accepted by DASA Automaton	C-46
3.2.1.3. 'SelectPhase' Function for DASA Automaton	C-52
3.2.2. Observations on the Definition	C-57
3.2.2.1. Manifestation of Desirable Properties	C-57
3.2.2.2. Nondeterminism in Definition	C-58
3.2.2.3. Explicit Appearance of Time	C-59
3.3. Scheduling Example Revisited	C-59
4. Analytic Results	C-61
4.1. Requirements for Scheduling Algorithms	C-61
4.2. Strategy for Demonstrating Requirement Satisfaction	C-62
4.3. Proofs of Properties	C-62
4.3.1. Algorithm Correctness	C-63
4.3.1.1. Proof: Selected Phases May Execute Immediately	C-63
4.3.2. Algorithm Value	C-64
4.3.2.1. LBESA Scheduling Automaton	C-64
4.3.2.2. DASA/ND Scheduling Automaton	C-68
4.3.2.3. Proof: If No Overloads, $c\{DASA\}$ and LBESA Are Equivalent	C-72
4.3.2.4. Proof: With Overloads, DASA May Exceed LBESA	C-76
4.3.3. Algorithm Tractability	C-106
4.3.3.1. Procedural Version of DASA	C-106
4.3.3.2. Proof: Procedural Version of DASA Is Polynomial in Space and Time	C-110
4.4. Notes on Algorithm	C-112
4.4.1. Unbounded Value Density Growth	C-112
4.4.2. Idle Intervals During Overload	C-113
4.4.3. Cleverness and System Dynamics	C-115
5. Simulation Results	C-117
5.1. Simulator Design and Implementation	C-117
5.1.1. Requirements	C-117
5.1.2. Design	C-118
5.1.2.1. Activities and the Activity Generator	C-118
5.1.2.2. Integrated Scheduler	C-119
5.1.3. Implementation	C-120
5.1.3.1. Approach: Build from Scratch or Adapt an Existing Simulator	C-120
5.1.3.2. Source of DASA Implementation	C-121
5.1.3.3. Single Scheduler for Simulation	C-121
5.1.3.4. Simulator Display Messages	C-121
5.1.3.5. Modifications	C-122
5.2. Evaluation of DASA Decisions	C-123
5.2.1. Methods of Evaluation	C-123
5.2.1.1. Execute Existing Applications	C-123
5.2.1.2. Modifying or Reimplementing Existing Applications	C-125
5.2.1.3. Modeling Existing Applications	C-125

5.2.1.4. Simulating the Execution of a Parameterized Application	C-125
5.2.2. Workload Selection	C-126
5.2.3. Examination of DASA Behavior	C-127
5.2.3.1. Workload Parameters and Metrics	C-127
5.2.3.2. Scheduler Performance Analysis	C-129
5.3. Interpreting Simulation Results for Specific Applications	C-138
5.3.1. Some Interesting Applications	C-138
5.3.1.1. Telephone Switching	C-139
5.3.1.2. Process Control: A Steel Mill	C-140
6. Related Work and Current Practice	C-143
6.1. Priority-Based Scheduling	C-143
6.2. Deadline-Based Scheduling	C-144
6.3. Other Related Work	C-146
Appendix A. The General Scheduling Automaton Framework	C-151
Appendix B. Derivation of DASA/ND Scheduling Automaton	C-157

List of Figures

Figure 1-1: Examples of Time-Value Functions	C-8
Figure 1-2: Execution Profiles for Priority and Deadline Schedulers	C-11
Figure 2-1: Format of Scheduler Events	C-20
Figure 2-2: An Observer Monitoring the Scheduler Interface	C-21
Figure 2-3: Scheduling Automaton Structure	C-22
Figure 2-4: Operation Types and Originators	C-27
Figure 2-5: State Components of General Scheduling Automaton Framework	C-31
Figure 2-6: Operations Accepted by General Scheduling Automaton	C-34
Figure 2-7: Organizations of Scheduling Functions	C-39
Figure 3-1: Simplified Procedural Definition of DASA Scheduling Algorithm	C-43
Figure 3-2: State Components of DASA Scheduling Automaton	C-45
Figure 3-3: 'RequestPhase' Operation Accepted by DASA Scheduling Automaton	C-47
Figure 3-4: Other Phase Operations Accepted by DASA Scheduling Automaton	C-48
Figure 3-5: Resource Operations Accepted by DASA Scheduling Automaton	C-49
Figure 3-6: Functional Form of DASA Algorithm	C-54
Figure 3-7: Execution Profiles for DASA Scheduler with and without Aborts	C-60
Figure 4-1: State Components of LBESA Scheduling Automaton	C-65
Figure 4-2: Operations Accepted by LBESA Scheduling Automaton	C-66
Figure 4-3: Functional Form of LBESA Algorithm	C-67
Figure 4-4: State Components of DASA/ND Scheduling Automaton	C-69
Figure 4-5: Operations Accepted by DASA/ND Scheduling Automaton	C-70
Figure 4-6: Functional Form of DASA/ND Algorithm	C-71
Figure 4-7: Histories Accepted by LBESA Beginning With $E_1 \cdot E_2 \cdot E_3$	C-105
Figure 4-8: Procedural Definition of DASA Scheduling Algorithm	C-107
Figure 5-1: Logical Structure of Simulator	C-119
Figure 5-2: Average Scheduler Performance with No Shared Resources	C-130
Figure 5-3: Average Scheduler Performance with One Shared Resource	C-131
Figure 5-4: Average Scheduler Performance with Five Shared Resources	C-132
Figure 5-5: Scheduler Performance Range with No Shared Resources	C-135
Figure 5-6: Scheduler Performance Range with One Shared Resource	C-136
Figure 5-7: Scheduler Performance Range with Five Shared Resources	C-137

Chapter 1

Introduction

This is a draft of a doctoral dissertation. The work presented is continuing. The material contained in this draft will be edited, and possibly augmented, to produce the final version of the dissertation.

Real-time applications are typically composed of a number of cooperating activities, each contributing toward the overall goals of the application. The physical system being controlled dictates that these activities must execute certain computations within specific time intervals. For instance, safe operating practices may dictate that an activity scheduled in response to an alarm condition must complete execution within several milliseconds of the receipt of the alarm signal.

Real-time applications usually contain more activities that must be executed than there are processors on which to execute them. Consequently, several activities must share a single processor, and the question of how to schedule the activities for any specific processor — that is, deciding which activity should be run next on the processor — must be answered. Necessarily, the prime concern in making scheduling decisions in real-time systems is satisfying the timing constraints placed on each individual activity, thereby satisfying the timing constraints placed on the entire application.

One factor significantly complicates the scheduling problem: the activities to be scheduled are not independent. Rather, the activities share data; execute mutually exclusive pieces of code, called critical sections [Peterson 85]; and send signals to one another. All of these interactions can be modeled as contention for resources that may not be shared. That is, once an activity has gained access to a shared resource, then no other activity can gain access to it until the first activity has released it. Any activity that is waiting for access to a resource currently held by another activity is said to *depend* on that activity, and a *dependency relationship* is said to exist between them. Dependency relationships are able to encompass both precedence constraints, which express acceptable execution orderings of activities, and resource conflicts, which result from multiple concurrent requests for shared resources.

No existing scheduling algorithm solves the problem of scheduling a number of activities with dynamic dependency relationships in a way that is suitable for real-time systems. This thesis addresses that problem. The resulting work provides an effective scheduling algorithm, a formal model to facilitate the analytic proof of properties of that algorithm, and simulation results that demonstrate the utility of the algorithm for real-time applications.

1.1. Problem Definition

A real-time system consists of a set of cooperating, sequential *activities*. These activities may be Mach threads [Mach 86], Alpha threads [Northcutt 87], UNIX processes [Ritchie 74], or any other abstraction known to the operating system that embodies action in a computer.

These activities interact by means of a set of shared *resources*. Examples of resources are: data objects, critical code sections, and signals. Any given resource may be used by only one activity at a time. If activity A_1 is accessing a resource when activity A_2 requests access to the same resource, A_2 must be denied access until A_1 has released the resource. Here, activity A_2 depends on activity A_1 since it cannot resume its execution until A_1 has released the resource.

We assume that activities can be preempted at any time. That is, at any time, the activity that is currently being executed by the processor may be suspended. Later, it may either be resumed or aborted, or it may never be executed again. If the activity is resumed, it will continue execution at the point at which it was interrupted. If it is aborted, the resources it holds will be returned to a consistent state and released.

Of course, the preemption of an executing activity, which is a manipulation of a computing abstraction, does not preempt the physical process that the activity is monitoring and controlling. Regardless of the execution state of the corresponding computer activity, the physical process continues to exist and, possibly, to change¹.

We also assume that scheduling decisions must be performed *on-line* — that is, they cannot be determined in advance due to the dynamic nature of the systems of interest. For instance, while the scheduler knows about the current activities, it does not know their resource requirements (that is, which resources will be needed, for how long, and in what order)². Furthermore, new activities may be created without warning — perhaps in response to external events. Since the set of activities to be scheduled may change over time, as may their dependency relationships, the scheduler must examine the activities to be scheduled in an on-line fashion.

[Ullman 75] demonstrated that the general preemptive scheduling problem is NP-complete, implying that tractable scheduling algorithms in even fairly simple systems cannot be optimal in all cases. Instead, they are designed to exhibit properties that seem likely to result in desirable behavior. As will be shown, our algorithm possesses a number of promising properties with respect to real-time systems.

¹Furthermore, the concept of an aborted computation is somewhat different in a real-time system than it is in other applications. In any setting, aborting an activity should result in returning the data items modified by that activity to a consistent state. However, in a real-time system not all of the actions of the activity are nullified by restoring consistent data values. Changes made in the physical world by means of computer-controlled actuators may have to be nullified. Opening a valve, for example, may have had an effect in the physical world that cannot be undone by simply closing the valve once again. In such cases, further compensatory actions may be required.

²For specific, restricted applications, it may be possible to know some or all of this information in advance; but, in general, it is impossible.

1.1.1. Dependencies

As defined earlier, activity A_1 depends on activity A_2 if activity A_1 cannot resume execution until A_2 has taken some action. For example:

1. several activities access a shared region of memory and access is arbitrated by a lock; when one activity holds the lock when another activity requests it, the requesting activity is blocked and depends on the first;
2. similarly, locks may be used to protect devices and sections of code; this allows the implementation of critical sections, for instance; in this case, whenever one activity is blocked while another activity is executing a critical section, the blocked activity depends on the executing activity;
3. precedence constraints, which impose partial orderings on the execution of activities, may be implemented by means of signals between activities; an activity that must complete a computation before another activity can begin, for instance, signals the second activity when it is done; the signal indicates that the second activity can resume execution; notice that the second activity depends on the first while it is blocked waiting on the arrival of the signal.

These dependencies clearly have an effect on scheduling. A number of activities may be blocked due to dependencies on other activities, but their resource needs are real and should be taken into account insofar as possible by the scheduler. However, in a typical operating system, if an activity is blocked, its requirements are not considered by the scheduler. As a result, important activities may be ignored by the scheduler. In particular, in a real-time system, activities that have pressing time constraints may be ignored because they are blocked due to dependency relationships.

A classic example of this type of behavior exists in the context of static priority scheduling systems [Peterson 85]. The most important activities are assigned high priorities, while less important activities are assigned low priorities³. Suppose that a low priority activity is executing a critical section when a new event makes a medium priority activity ready to run. A priority scheduler would preempt the low priority activity immediately, while it was still executing its critical code⁴. If a high priority activity subsequently became ready to run, it would preempt the medium priority activity. Unfortunately, if the high priority activity were to attempt to execute a critical code section, it would be blocked and the medium priority activity would resume execution regardless of the relative urgency of their respective time constraints.

Another example, similar to the one just presented, will be examined more closely in a later section of this thesis.

A model that keeps track of blocked activities and the reason that each activity was suspended can cover all of the scenarios that have been mentioned so far. Specifically, activities that share data can coordinate

³Note that there is no inherent correlation between an activity's priority and the urgency of its time constraint. This is a key problem with static priority schedulers.

⁴Some systems prevent preemption at these times, while many do not [KB 84, Bach 86]. But even systems that prevent preemption suffer from other problems— for example, they have longer, potentially unbounded, response times, and they lose information by describing an activity by a single number, its priority. This latter point will be elaborated in later sections of this document.

access to that data by means of a lock manager. If a lock request is granted, then the data may be accessed. If a lock cannot be granted immediately, the requesting activity is blocked and becomes dependent on the activity currently holding the lock.

In the case of critical sections, permission to execute a critical section can be arbitrated by semaphores. When an activity executes a *P* operation to request permission to execute a critical section, the activity either begins executing the critical section immediately, or it is blocked because another activity is already executing that critical section. In the latter case, the blocked activity is dependent on the completion of the activity executing its critical section. Similar dependencies also result from more general uses of semaphores.

Finally, signals between activities can often be implemented using a semaphore. The signal originator issues a *V* operation, enabling the signal receiver to continue execution when it does a *P* operation to detect whether the signal has been sent yet. If the *V* precedes the *P*, then the signal was sent before the receiver looked for it, and the receiver is allowed to continue. Otherwise, the signal receiver must wait until the signaller has issued the signal. At that time, the receiver is blocked and its further execution depends on the continued progress of the activity that will send the signal.

In each case, the scheduler can acquire the information it needs to construct a complete picture of the dependencies in the system. Conversely, since this information is available, this thesis applies to a wide range of applications and systems in which these types of dependencies occur.

1.1.2. Real-Time Systems

Despite common definitions that refer to artifacts such as interruptability in the kernel, interrupt latency, and context swap times [Rauch-Hindin 87], real-time systems are fundamentally concerned with carrying out activities according to timing constraints imposed by an application — that is, the external world. The timing constraints imposed by the external world imply that the time at which an activity is performed is just as important as the correctness of the computation being performed. Note that a faster computer that executes activities in an unfortunate order might be less "real-time" than a slower computer that executes the activities in a more advantageous order.

There are several classes of real-time systems [Bennett 88]. *Low-level* real-time systems are typified by loop control applications, where computers interrogate sensors, perform a fixed set of calculations on the sampled data, along with other state information, and control a group of actuators based on the results of the calculations. The activities that implement these applications are often executed periodically — sometimes because the sensors produce data periodically (e.g., radar) and sometimes because the control models on which the systems are based require periodicity.

Often, several of these low-level real-time systems are monitored and controlled by a higher level real-time system, called a *supervisory control system*. For supervisory control systems, the application events that trigger activity are typically not periodic; rather, they occur stochastically — for example, in

response to an alarm condition or to indicate the completion of a low-level sequence of operations. These events represent significant changes in the physical world and must be handled by the supervisory control system in a timely manner. So, just like low-level real-time systems, supervisory control systems have physically derived time constraints; and, in fact, meeting these time constraints is just as critical as it is in low-level systems.

In addition to monitoring and directing the low-level real-time systems, supervisory control systems perform strategic planning functions — that is, they determine how to coordinate the actions of the lower-level systems to meet the application's objectives — and they receive direction from higher level management information systems. Typically, this information would include the specific objectives for the supervisory control application (for example, to produce the goods that fill a given set of orders during the current shift). Although supervisory control activities cooperate to provide their services, they still contend for access to shared system and application resources.

Unfortunately, the policies that are prevalent in non-real-time systems to resolve such contention are inappropriate, and may in fact be counterproductive, in real-time systems. For instance, in time-sharing systems, fairness is desired and is obtained by, among other things, using FIFO queue disciplines and round-robin schedulers [Peterson 85]. This approach reflects the belief that all activities are equally significant. However, in real-time systems this is clearly not the case — some activities, and hence, some time constraints, are decidedly more significant than others. In fact, while failing to satisfy some time constraints may have no adverse effect on the physical process or platform being controlled, failing to satisfy others can have catastrophic effects. A few examples will illustrate the varying significance that may be attached to meeting specific time constraints.

First of all, consider a real-time supervisory control system in a process control setting — a furnace and a continuous caster in a steel mill. Molten steel of a specific chemistry is created from iron, scrap, and additional materials in the furnace. When the metal in the furnace is ready to be converted into slabs of solid steel, the molten metal is poured into a large ladle, transported to the caster, poured into the caster, and cast into a long, continuous slab that is subsequently cut into individual slabs of appropriate length. When the metal is originally poured into the caster's "mold," it is liquid. It cools in the "mold" and is solid when it emerges, ready to be cut. Several low-level real-time systems directly control the furnace, the caster, and several related pieces of equipment. These systems are monitored, controlled and coordinated by a supervisory control system.

In this setting, there are several types of supervisory control time constraints that can be examined. Roughly speaking, they fall into three classes: (a) time constraints that, if missed, will result in potential loss of life and property (e.g., due to liquid steel spilling over the area); (b) time constraints imposed by the physical world that have financial penalties if they are missed (e.g., losing quality control statistics for products, resulting in potentially unusable products); and (c) time constraints that are not physically based and result only in inconvenience if they are missed (e.g., operator display requests).

Military systems also provide examples of the difference in importance between various time constraints.

For a fighter plane, for instance, the most importance activities are those that serve to keep the plane in the air and the pilot alive; the activities that control weapons are less important, although, obviously, they are still of great concern. On the other hand, aboard a ship, which will float stably without constant control, the activities in charge of the defensive weapons systems may well be more important than those that steer the ship.

The preceding examples demonstrate that there are a number of time constraints that an application declares and that there are significant differences in kind among the activities expressing those time constraints. It makes sense to talk about failing to satisfy time constraints in a dynamic system because transient, and even permanent, increases in resource demands are possible. Some difficult questions, then, involve detecting these demand peaks and deciding which time constraints should be satisfied and which should not.

One final, critical observation should be made. Notice in the examples above that, although each activity was operating under a time constraint, there was a classification of its relative importance (compared to other activities) that was independent of the time constraint. That is, there was no inherent correlation between the activity's urgency, which was captured by its time constraint, and its importance. A critically important activity may require little computation time and may have a very loose time constraint (relatively speaking). In that case, it is certainly not an urgent activity, although it is an important activity. Conversely, a relatively unimportant activity may have a time constraint that is very tight. Therefore, it is fairly urgent even though it is not very important in the global scheme of things. Many schedulers are able to deal with an activity's importance (e.g., priority schedulers [Peterson 85]) or its urgency (e.g., deadline schedulers [Conway 67]), but few attempt to distinguish between these two attributes or to use all of the information that is captured in both of them.

1.2. Simple and Complex Schedulers

Two distinct approaches may be taken in designing and constructing a scheduler. On one hand, a minimal scheduler can be provided. The scheduling may be list-driven, like the rate group schedulers used by cyclic executives [GD 80, Stadick 83, MacLaren 80]; or it may employ a very simple algorithm, like a priority scheduler. Such approaches impose a low system overhead. This may be entirely appropriate when the goal is to maximize system throughput or to support a simple application structure so that properties (such as worst case load behavior) can be demonstrated, but it is not obviously the best approach for systems where the goal is to satisfy as many time constraints as possible or obtain the highest application-specified value as possible. Furthermore, minimal schedulers may have limited applicability, as evidenced by the fact that they are already stretched to the limit in large, dynamic real-time applications.

Alternatively, a complex scheduler may be used. In this case, application activities tell the scheduler their individual needs and the scheduler attempts to satisfy them, making decisions based on global information that the application does not possess. The more complete and accurate the information, the better the job

that the scheduler can do in managing resources⁵; and processor cycles, of course, are one particularly important resource.

This thesis explores the latter philosophy by allowing the scheduler to use more information than usual in order to do a better job of scheduling for real-time systems. There are two major points that must be demonstrated to verify the quality of the scheduling: first of all, the individual scheduling decisions must be good (i.e., show that the "right" activity was selected for execution); and secondly, the performance penalty paid for employing a more expensive scheduling algorithm must be more than offset by improved scheduling from the point of view of the application (i.e., show that the scheduler can make better use of the resources required to make the scheduling decisions than the application can). Of course, not every application requires a complex scheduler, but some do, and this thesis explores the use of complex schedulers to support those applications.

The previous discussion has focused on *time constraints* without elaborating on the precise definition of these constraints. The term has deliberately been used to capture the general notion that real-time computations must satisfy certain timing requirements. We now introduce a formal method to describe time constraints and introduce some additional terminology. Each activity in a real-time application is composed of a sequence of disjoint *computational phases*, also known simply as *phases*. The application as a whole makes progress when its component activities make progress; and each activity makes progress by completing its computational phases. Therefore, the completion of a computational phase marks measurable progress for the application, and this progress is expressed in terms of *value* units. Associated with each phase, then, is a *time-value function* [Jensen 75] that specifies that phase's time constraint — it indicates the value acquired by the application for completing the phase as a function of time.

The shape of the time-value function is arbitrary, and Figure 1-1 shows a few examples. Figure 1-1(a) shows a step function of height v . In this case, completing the computational phase by time t_{dl} yields value v , while completing it at any later time yields no value. Figure 1-1(b) shows a situation where the cutoff in value is not as sharp. Prior to time t_c , the value associated with completing the computation is again v . However, following that time, the value decreases smoothly until, once again, a point is reached after which no value is gained by completing the phase. Finally, Figure 1-1(c) corresponds to a phase that must complete within a certain interval in order to acquire a non-zero value for the application. Although sharp transitions are shown at both t_{c1} and t_{c2} , more gradual transitions — such as a parabola — could also be used. Finally, the times at which there are sharp changes in time-value functions are known as *critical times*. Times t_{dl} , t_c , t_{c1} , and t_{c2} are all critical times.

The simple step function shown in Figure 1-1(a) illustrates several key ideas and allows the introduction of some important terminology. First of all, time t_{dl} is referred to as a *deadline* since it represents the last instant at which the phase can complete and still make a non-trivial contribution to the accrued value for

⁵Improved scheduling can also be obtained by devoting more resources to analyzing a fixed amount of scheduling information. Although the main thrust of this thesis is to study the use of more information than usual, the algorithm to be studied also requires significant resources for the scheduler. The resulting implications will be discussed later in the document.

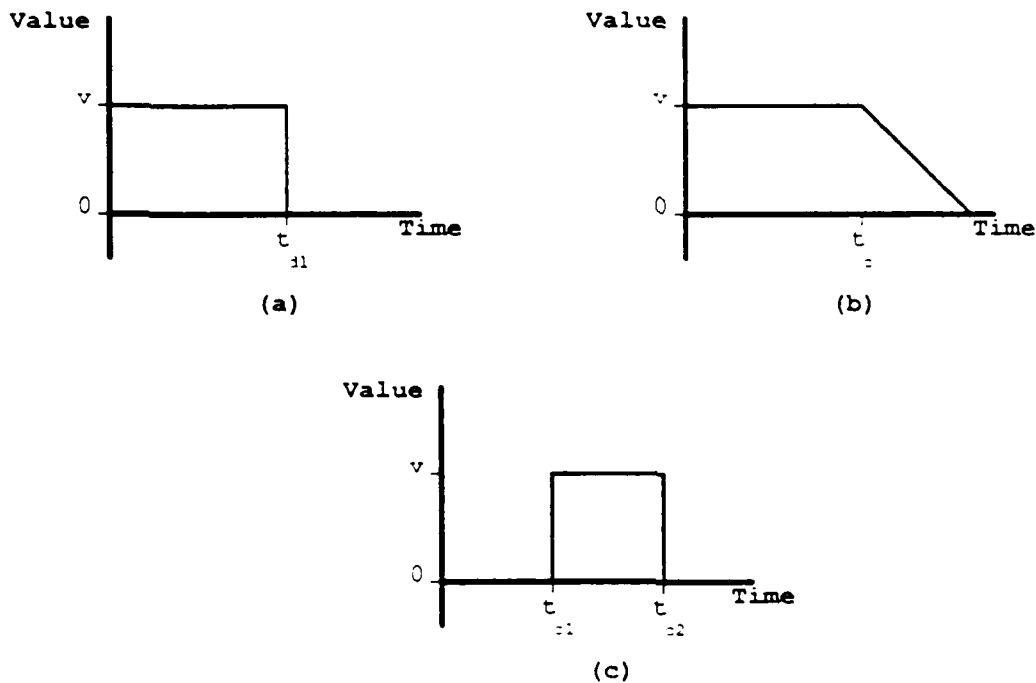


Figure 1-1: Examples of Time-Value Functions

the application. Value v is called the *importance* of the phase. If every time-value function were a step-function and all of the step functions had the same height (importance), then each phase that was completed would make an identical contribution to the progress of the application and an appropriate scheduling strategy would complete as many phases as possible prior to their respective deadlines. If, however, different phases were to have different importances, then they would make different contributions to the value accrued by the application and the scheduling strategy that would maximize that value would be different. Considered over the lifetime of an application, a greater accrued value represents a more successful application.

If resource demands, including those for processor cycles, are sufficiently low, then all activities can be scheduled, thereby accruing a large value for the application. However, in the event that it is impossible to satisfy all of the activities' resource demands, an *overload* exists. In this case, some subset of the activities will meet their time constraints, while others will not, resulting in a lower accrued value for the application. In an overload situation, the scheduler should maximize the value accrued by the application.

With an understanding of the simple step function time-value function and the vocabulary introduced above, consider again the notion that a scheduler can do a more effective job when it has more complete or better quality information on which to base decisions. Consider the algorithms a scheduler can use given specific types of information (unless otherwise noted, these are all discussed in [Conway 67], [Janson 85] or [Peterson 85]):

- no information — there is no way to distinguish activities so round-robin or random scheduling of ready activities would be appropriate;
- relative importance of activities — priority scheduling of ready activities; this algorithm would always run the highest priority (most important) ready activity;
- deadline and required computation time of activities — deadline scheduling, where the ready activity with the nearest deadline is always selected to run, or slack-time scheduling, where the ready activity that has the least slack-time⁶ is always selected to run, would be optimal algorithms with this information;
- time-value functions [Jensen 75], which capture importance and timing requirements — more complex schemes such as best-effort scheduling [Locke 86] of ready activities can be employed; Locke showed that under his model, this approach can be more effective than those listed above.

This thesis will explore the consequences of allowing the scheduler to have access to not only the activities' time-value functions, but also to information describing the dependency relationships existing between activities. This should enable the system to take into account the time constraints of blocked activities, allowing a better ordering of activities, along with the earlier detection and better resolution of overloads.

Notice that the dependency information that is to be used by the proposed scheduling algorithm is not very exotic or difficult to obtain in many cases. Often, the operating system or a system utility, such as a lock manager, holds key pieces of this information. Whenever an activity is unable to gain immediate access to a shared resource, it is typically blocked. At that point, the system is capable of noting which resource is being accessed, as well as the identities of the activities holding and requesting the resource. In other cases, straightforward extensions to the operating system interface would provide the necessary dependency information for the scheduler's use. As a result, if the algorithm can be demonstrated to have sufficient merit, an implementation would not seem to be unduly difficult.

1.3. Scheduling Example

In order to demonstrate some of the points that have been made earlier and to illustrate the type of problem that is to be addressed by this thesis, consider an example.

Assume that there are only three activities, each consisting of only a single phase. Designate these phases p_a , p_b , and p_c . Phase p_a has a relatively low importance, requires four time units of execution time to complete, and must complete execution within 15 time units of its initiation. It requires the use of shared resource r . It requests access to r after it has executed for one time unit, and releases r after it has executed for a total of three time units.

Phase p_b has a medium importance, requires three time units of execution time, and must complete within four time units of its initiation. It also uses shared resource r . Like p_a , it requests r after it has executed for one time unit and releases it after it has executed for a total of three time units.

⁶slack-time = deadline - present time - required computation time.

Phase p_c has a relatively high importance, requires four time units to complete execution, and must complete within ten time units of its initiation. It does not access shared resource r .

All of these phases are initiated as a result of external events. Suppose that the event that initiates phase p_a occurs at time $t = 0$, and the event that initiates both p_b and p_c occurs two time units later. This implies that the deadline for completing phase p_a is time $t = 15$, the deadline for completing phase p_b is at time $t = 6$, and the deadline for completing phase p_c is time $t = 12$.

If these phases are to be scheduled using a priority scheduler, then it seems clear that their importance to the application should act as an indication of their priority. Therefore, if $Pri()$ is a function that returns the priority of a phase . . .

$$Pri(p_a) < Pri(p_b) < Pri(p_c)$$

Also notice that this is a situation where urgency, when defined as the nearness of a deadline, is not the same as importance. To see this, let $DL()$ represent a function that returns the deadline of a phase. Then

$$DL(p_b) < DL(p_c) < DL(p_a)$$

A priority scheduler will always execute the ready phase with the highest priority. A deadline scheduler will always execute the ready phase with the nearest deadline. Whenever a phase is waiting on a resource, it is blocked and so is not ready. Applying these rules to phases p_a , p_b , and p_c yields the execution profiles shown in Figure 1-2. The x-axis represents time, while the y-axis indicates which phase is executing at any given time. Significant events in the executions of the phases are indicated. Notice that neither the priority scheduler or the deadline scheduler could meet all three deadlines. Both failed to allow phase p_b to meet its deadline. A more sophisticated version of the priority scheduler, for example one of the priority inheritance schedulers mentioned earlier, will not solve the problem either. The algorithms to be investigated in this thesis will solve this problem.

1.4. Motivation for the Model

Much of the model of supervisory control systems that has been presented is straightforward and is largely based on current practices and systems. Nonetheless, a few points — most notably the use of application-specific values within the system — may not be obvious or typical of existing implementations. These issues will be further explained in the following sections.

1.4.1. Accrued Value

Evaluating a scheduling algorithm by determining the total value it accrues while executing an application is unusual. However, not only is it intuitively appealing, it is also appropriate in many cases.

The intuitive appeal lies in the view that accumulating value represents making progress. As each activity completes designated portions of its execution, value accrues to indicate the utility to the application of that particular computation.

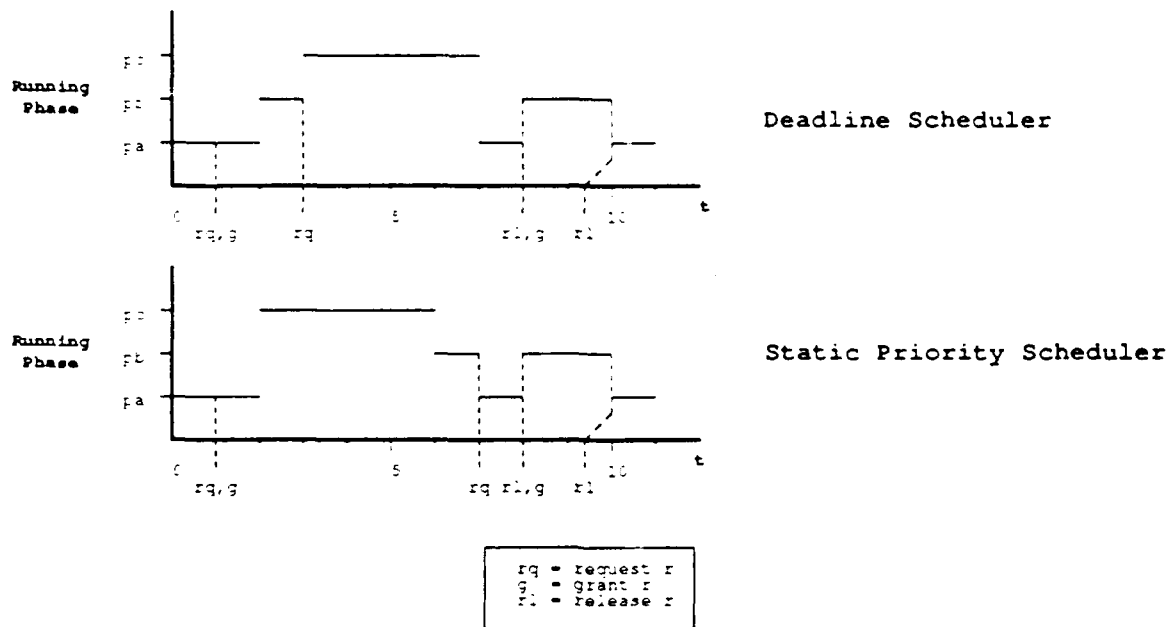


Figure 1-2: Execution Profiles for Priority and Deadline Schedulers

While this might sound plausible as a metric, there remains the question of whether values can be assigned meaningfully to computational phases of an activity. In many instances, there is strong reason to believe that this is the case.

The class of process control applications provides one example of the applicability of this approach. Typically, one or more processes are being controlled or one or more products are being manufactured under the supervision of a single supervisory control computer system. Since the goods being produced have a monetary value, it is possible to assign values to particular activities based on the commercial worth of the goods being produced by each activity. Consequently, the use of a scheduler that maximizes the amount of value accrued for the application is actually maximizing the commercial value of the goods being produced. This seems entirely reasonable. (Conversely, if it seemed more natural, the notion of monetary loss or penalty could be used instead of the monetary value or profit outlined. The underlying notion is essentially the same in either case.)⁷

During an overload, when there are insufficient resources to meet the overall demand, some activities

⁷The use of monetary measures to determine schedules has long been used in the operations research and job shop scheduling communities. The model used in this work differs somewhat from their model. This is dealt with in some depth in Chapter 6. Briefly, the typical job shop model assumes that the set of orders currently known will all be filled at some point in time. That is, all activities will eventually be run. This does not take into account the fact that in real-time computer systems, some activities are of only transient value because they are run frequently or because they must be run in a timely fashion or not at all due to the quality of the information or the physical time constraints of the application.

may not be scheduled. It would be perfectly reasonable to select which of two activities should be run based on their relative values. In fact, it would be possible that during an overload involving three or more activities, the activity with the highest individual value would not be scheduled. Rather, two or more activities with lower individual values, but with a higher combined value, could be scheduled.

This overload behavior should be contrasted with that of other scheduling policies. For instance, a priority scheduler would always execute the activity with the highest individual value at any given time (assuming that the priorities assigned to activities corresponded to the commercial worth of the activity as described previously). In the case just outlined, this would result in a lower total value than the method that maximized value.

A steel mill application can illustrate this point, while demonstrating the dynamic nature of the assignment of values to tasks. The steel mill under consideration has a furnace and caster that combine to transform raw materials into slabs of finished steel of specified chemistry. There are two functions that are particularly interesting: chemistry control, which controls the chemical composition of the steel being produced, and quality control tracking, which follows the progress of the steel through various stations in the mill including the caster and associates a specific chemistry with each foot of every steel slab produced by the mill. A single supervisory control computer monitors and controls both of these functions.

During overloads, the supervisory computer may have to decide which function should be run. Most often, the value associated with the quality control activity should be higher than that associated with the chemistry control activity. This is because it is important to know what is in each steel slab that is sold. In fact, since many customers will not buy a slab without detailed knowledge of its chemistry, the profit that would be realized from the slab is at stake if the tracking activity does not execute in time. On the other hand, if the chemistry control activity is not executed, the chemistry of the steel may be different from what was intended. This is acceptable if the resultant chemistry is one that can be sold or can be further processed to obtain such a chemistry. Notice that the chemistry — even if it is not the chemistry that was originally intended — is known and can be tracked by the quality control activity.

The dynamic nature of value assignments is shown by the fact that the above generalization does not hold in every case. When a particularly rare chemistry is desired, it is sometimes the case that the steel cannot be sold if the chemistry is not exactly right, therefore placing the profit for the heat in jeopardy if the chemistry control activity is not run. It is possible that the profit involved, especially for a specialty steel, will outweigh the profit that will result from tracking steel slabs of more typical chemistries through the rest of the mill. Since these decisions vary with each heat (mix) of steel, values must be assigned to the chemistry control and quality control tracking activities dynamically to correspond to each heat.

Military defense systems are a second class of applications that seem to allow values to be assigned to component activities meaningfully and would benefit by using a scheduler that maximized accrued value for the application. In this case, the value accrued for an activity controlling a defense system would be derived from the number of lives or the number of other military assets that can be saved. As unsettling as it is to consider, it seems wise to employ a scheduler that maximizes the number of lives or assets that are successfully defended.

These examples make use of the fact that there is a common "currency" in which values can be expressed naturally — money in process control situations and lives or other military assets in combat systems. In such situations, it is relatively straightforward to assign values to various activities⁸. Other applications may require that values take into account a number of different factors — money, lives, operator satisfaction, and so forth — and appropriate weightings of these factors will have to be developed to produce acceptable and meaningful activity values.

Of course, the real test of the utility of this approach will come in the future when scheduling algorithms that maximize application-defined value are employed in production systems — or, perhaps, prototype versions of production systems. At that time, the performance of these systems can be compared directly to alternative approaches. Pending the outcome of such tests, it does seem to be useful to explore the notion of maximizing the value for an application.

1.4.2. Time-Value Functions

As shown in the above discussion, the notion of assigning values to application activities and scheduling activities to maximize the accrued value for the entire application has merit in a wide range of applications. These assigned values reflect the relative importances of the activities that they represent.

Since the systems under consideration for this work are real-time systems, the value associated with the completion of a computation varies as a function of time. For example, in an automated assembly application, the value of closing a mechanical manipulator to grasp a part on an assembly line is a function of time. If the grasping motion is completed too soon, the part will not have reached the manipulator yet. If the grasping motion is completed too late, the part will have already passed by the manipulator.

Time-value functions facilitate the description of the time constraints and relative importances of the activities comprising a real-time application. The time-value function records the value to be accrued by completing the designated computational phase at each point in time.

Time-value functions seem to be a fairly natural expression of the utility of completing a given computation as a function of time in many situations. A skilled operator in a process control environment or a carefully constructed functional requirements document for the system will often be capable of describing all of the information encoded in a time-value function.

Although time-value functions are a relatively new formalism for expressing the relative urgency and

⁸This act of assigning values to specific activities comprising an application corresponds roughly to the normal assignment of priorities to activities (where the activities are often called processes or tasks). In many modern applications a number of activities coordinate to provide a single application-level logical function, such as material tracking. In such systems, some activities may provide a specific service, such as accessing a tracking database, to a number of other activities with widely varying values. The assignment of a single value to the server activity is difficult. If it has a lower value than the activity that it is currently serving, then it may not be scheduled as quickly as it should be. On the other hand, if it has a higher value than the activity it is serving, then it may consume resources that could, and should, be used by other activities. This problem is alleviated if an approach is taken where the activities in the computer application can correspond directly to the application-level logical functions, while still providing for modular construction of the application. This has been done in the Alpha Operating System.

importance of each activity in a real-time system, they are beginning to make the transition into practice and have been used successfully in a few selected contexts.

1.5. Technical Approach

The technical approach described in this section has been adopted in order to carefully address the problem of scheduling with dependencies and to explore and evaluate potential solutions. Briefly, the approach consists of the following major steps:

1. define a computational model within which to work;
2. devise an algorithm that possesses the required properties and express it within the computational model;
3. insofar as possible, demonstrate analytically the correctness, utility, and tractability of the algorithm;
4. simulate the performance of the algorithm on common classes of supervisory control systems and compare with other relevant algorithms or ideals.

1.5.1. Define Model

The first step, defining a computational model, is intended to provide a clear, useful framework that will capture the essential aspects of the problem to be solved and will also support the specification of unambiguous solutions, embodied primarily as scheduling algorithms. The need for a model that exhibits all of the desired problem features, while excluding all factors that are non-essential for the problem statement and solution is obvious. If the work is done with the simplest model that accurately expresses the problem, then the work will be more comprehensible and succinct. Equally important is the requirement that the model support the unambiguous specification of scheduling algorithms. Without such definitions, the ability to perform precise/definitive analytic proofs to demonstrate properties of an algorithm will be lost. Also, a set of requirements for problem solutions is formulated in terms of the computational model.

1.5.2. Devise Algorithms

After the model has been created, it is possible to begin exploring various algorithms within the framework provided by the model. While the computational model is intended to support the development of a number of scheduling algorithms and will provide an excellent platform for the extension of this work in the future, this thesis does not explore a wide range of alternative algorithms exhaustively. Rather, it identifies and characterizes the behavior and performance of a single algorithm that has the desired properties, called the *Dependent Activity Scheduling Algorithm* (DASA). This algorithm will be described in two forms — a formal, mathematical form that will be used to define the algorithm and to support analytic proofs and a procedural form to provide a measure of the algorithm's complexity and to support the simulation work that has been done.⁹

⁹Actually, the mathematical definition features non-determinism in certain places, indicating that ordering is unimportant with respect to the algorithm at those points. The procedural definition, however, does not contain any non-determinism and so can be viewed as a single specific implementation of the algorithm that the mathematical definition describes.

1.5.3. Prove Properties Analytically

Once the DASA algorithm has been defined, analytic proofs that demonstrate that it satisfies the problem requirements may be devised. The formal model that is used to describe the scheduling algorithms is based on automata that accept certain sequences of scheduling events. There is a different automaton associated with each distinct scheduling algorithm. So, for example, the automaton associated with the DASA algorithm will accept any sequence of scheduling events that is consistent with the behavior of the DASA algorithm. Such automata can also accumulate the value assigned to an execution history. By comparing the execution histories accepted by the automata corresponding to different scheduling algorithms, proofs can be constructed that show that two scheduling algorithms accept different histories. Furthermore, the proofs may compare the values accumulated for all of the execution histories accepted by the automata representing certain scheduling algorithms for a specific set of phases with specific time-value functions and computation time requirements. (Taken together, these last two items — a phase's time-value function and its computation time requirement — are referred to as the phase's *scheduling parameters*.) Such comparisons can be used to demonstrate that one scheduling algorithm is capable of generating schedules that are superior to those of another algorithm, measured in terms of total value accrued by the application during its execution history.

Unfortunately, real-time systems featuring complex, dynamic dependency relationships are quite complex. And, although the analytic proofs can make some observations about the correctness, behavior, and value of the algorithm, a complete case for its utility cannot be made without demonstrating its performance under realistic conditions. To address this need, simulations have been carried out to investigate the performance of the DASA algorithm and to demonstrate properties that cannot be proven analytically.

1.5.4. Simulate Algorithm

A parameterized workload has been devised that can mimic various numbers of activities displaying a range of access patterns to a set of shared resources. Using this workload, a suite of simulations has been run. These simulations compare the benefit of using the DASA algorithm instead of a more standard algorithm — for instance, a static priority or deadline scheduling algorithm with FIFO queueing for access to each shared resource. They also compare DASA's performance with a reasonable estimate of the theoretical maximum value that can be obtained. The DASA scheduling algorithm is relatively complex when compared to more standard scheduling algorithms. Consequently, in a uniprocessor implementation of the algorithm, DASA will require more time to select an activity to execute than a more standard algorithm would. In order to be fair in performing comparisons among scheduling algorithms, this additional overhead is also taken into account. The simulation results reveal situations in which applying the DASA algorithm will probably be profitable.

Chapter 2

The Scheduling Model

Models are central to abstract study. They allow the salient features of a potentially complex system to be isolated and restrict the size of the space of possibilities to be investigated. Properly specified, a model provides an unambiguous definition of the behavior of a system and highlights the underlying assumptions that are made by the investigator. Within the framework of the model, simulations and analytic analyses may be performed.

To take advantage of all of these properties, a model has been devised that possesses the necessary richness and within which scheduling algorithms can be studied. This chapter presents this model and describes the rationale that shaped it.

A formal computational model has been constructed to facilitate the definition and formal analysis of scheduling algorithms. Initially, this model is presented informally in order to allow for a natural discussion of the issues that shape the model and the intended structure of the model and the environment provided by real-time applications. This is followed with a formal description that provides a detailed, precise specification of the model.

2.1. Informal Model and Rationale

The informal discussion of the computational model will describe each of the principal elements of the model in general terms. This should allow the reader to have an intuitive grasp of the interplay of various elements of the model without having to wade through a mass of symbols and mathematics. This will set the stage for the presentation of the formal model, where all of the details will be specified for each of the principal elements of the model.

2.1.1. Applications, Activities, and Phases

As mentioned in the previous chapter (in Sections 1.1 and 1.2), an application is composed of a set of activities. Each activity, in turn, comprises a sequence of computational phases, and each computational phase is characterized by a time-value function that indicates the importance and urgency of that phase. At any given time, an activity is operating in a single computational phase so that the activity can be uniquely identified by designating the phase that is currently underway. Therefore, the complete set of activities can

always be represented by the set of phases currently in progress¹⁰, and this set can be designated as:

$$\{p_0, p_1, p_2, \dots\}$$

The execution of an application involves sharing the single processor among the set of active phases over time. The determination of which phase to run at any given time is made by the scheduler, one of the major components of the operating system, based on the relevant information available to it.

2.1.2. Shared Resources

Phases may access shared resources. A request for such access is signalled by a phase by means of a 'request' event for the specific resource desired. Permission to access a shared resource is given to the phase by means of a 'grant' event.

All shared resources that are held by an activity must be released at the completion or abortion of each computational phase. This assumption is justifiable on two counts, but may, at the same time, seem to be restrictive. First of all, when a phase represents a distinct logical stage in a computation, there is good reason for expecting that the resources used to carry out that phase may be released upon its completion. Of course, if phases are used to represent very fine grained portions of a computation, then this assumption may be called into question. However, since each phase is a unit of computation that corresponds to a single time-value function, and since the time constraints that dictate the time-value functions are derived by the physical necessity of completing a computation in a certain time frame, it seems clear that using phases to delimit very small portions of an activity departs from the expected, and useful, application of phases to decompose activities in a real-time system.

The existence of stylized applications or system facilities gives rise to the second justification for the assumption that all shared resources are released at the completion of a computational phase. One specific example of such an application is an atomic transaction facility. The use of transactions in real-time systems is appealing, but the question of how to schedule them is unsolved. By allowing this model to capture the behavior of transaction facilities as well as the assumed normal behavior of real-time activities, the work presented here can hopefully make a somewhat greater contribution.

2.1.3. Phase Preemption

At any given time there is one phase that is actively executing on the processor. That phase may be preempted by the scheduler at any time. A preemption is signalled by a 'preempt-phase' event. Should the scheduler subsequently determine that the phase should be resumed, it would issue a 'resume-phase' event.

¹⁰For the purposes of this model, a phase is considered to be "in progress" as soon as it is made known to the operating system. So, for instance, a phase that has never executed a single instruction of its code is nonetheless considered to be in progress — it has progressed far enough to submit its initial resource request (in terms of required processing time, importance, and urgency) to the system.

2.1.4. Phase Abortion

The scheduler may decide to abort a computational phase at any time. This is indicated by issuing an 'abort-phase' event for the phase to be aborted. A phase might be aborted to free a shared resource more quickly than it would otherwise be freed. Or, a transaction facility might issue an abort in response to a component failure or to resolve a detected deadlock.

The amount of time required to completely process an abort depends on the number and type of resources held by the phase being aborted. Each time access to a new shared resource is granted to a phase, the amount of time required to abort the phase is incremented by an amount dependent on the newly granted resource.

The incremental amount of abort time associated with a resource may arise from several sources. For instance, for resources that are treated like data objects in a traditional database system, each data object altered during the course of an aborted transaction must be returned to the same state it had prior to the transaction. The time required to restore this pre-transaction state is determined by the time required to find the desired value followed by the time required to actually update the data object.

In other cases, more must be done than merely restoring the state of the appropriate memory locations. Real-time systems often control physical processes by regulating actuators that effect changes in the physical environment. Permission to manipulate an actuator may be acquired by successfully requesting exclusive access to a shared resource that is logically associated with the actuator. Once access to the resource has been granted, the actuator is available to, and manipulated by, the requesting computational phase. If the phase is subsequently aborted during its execution, then it is quite possible that the actuator may have to be manipulated once more in order to return the physical environment to an acceptable state. The amount of time required for such compensating actions must be included in the time allotted for abort processing for each resource of this type.

Following the completion of an abort, the effected activity will be ready to reexecute the aborted phase if time and resources permit.

2.1.5. Events

To motivate the development of a formal model, imagine that all of the major components of an operating system interact by signaling specific events to one another. Conceptually, these *events* encapsulate information and commands, and they can originate within the operating system or from the computational phases comprising the application.

As shown in Figure 2-1, each event includes an event timestamp, an *operation* name, appropriate arguments for the operation, and the originator of the event. Timestamps are used to provide a global ordering of all scheduling events. There are a small number of scheduler-related operations, which will be described below. And, as far as scheduling-related events are concerned, the originator of an event is either

the scheduler itself (meaning that the event passed across the interface from the scheduler to the rest of the operating system, possibly continuing on to an application phase) or an individual phase (meaning that the event passed from that phase to the scheduler, via the operating system).

	t_{event}	$op(parms)$	O
where,	t	is a timestamp,	
	op	is a scheduling operation (as defined in Fig. 2-4),	
	$parms$	is the set of arguments for the operation op ,	
	O	is the originator of the event (either p , for a phase, or S , for the scheduler)	

Figure 2-1: Format of Scheduler Events

2.1.6. Histories

Given this model of operating system structure, an observer located within the operating system could watch an application execute and monitor the interface between the scheduler and the rest of the operating system. (See Figure 2-2.) The observer could then record a sequence of timestamped events passing across the interface. Conceptually, these events would represent the communication of information and commands to and from individual activity phases and the scheduler.

Such a sequence of scheduling events is called a *history*. In general, any sequence of scheduling events constitutes a history, although not all histories are meaningful. To aid in recognizing which histories are potentially meaningful, definitions have been developed for well-formed histories (e.g., timestamps increase throughout the history, the only event operations included in the history are those listed in Figure 2-4) and for legal histories (i.e., well-formed where the sequence of events is plausible, for example, 'request's precede 'grant's). Operations on histories have also been defined to facilitate their manipulation. For simplicity, the only histories that are ever dealt with in formal analysis, after the introduction of these definitions, are legal, well-formed histories. (The definitions referred to in this paragraph are presented in Section 2.3.2.8.)

Different schedulers will select different activities for execution based on the relevant scheduling parameters for each phase under consideration. Consequently, different histories will be generated by different schedulers, even though they may be executing the same application under the same conditions. Examining these histories allows the performance and behavior of the schedulers to be compared and contrasted. Formally, the histories are examined by a special type of finite state automaton, called a scheduling automaton.

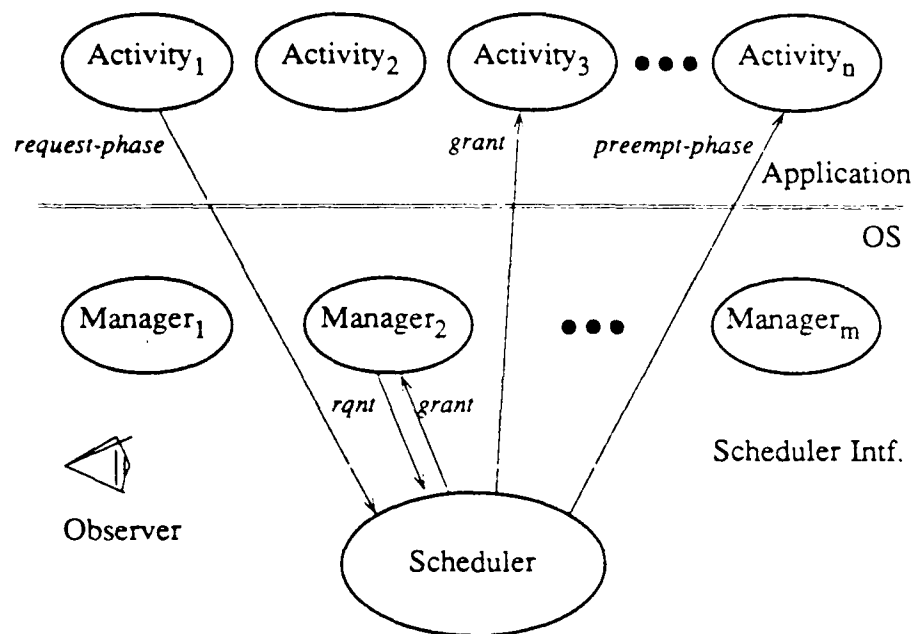


Figure 2-2: An Observer Monitoring the Scheduler Interface

2.1.7. Scheduling Automata

Since events and histories have been defined formally, automata can be created that recognize legal histories corresponding to various scheduling algorithms. Such an automaton is called a *scheduling automaton*.

Each scheduling automaton incorporates a scheduling algorithm. The automaton accepts — that is, recognizes — any history that could have resulted from the use of the scheduling algorithm that it embodies. All other histories contain some sequence of scheduling events that could not possibly have resulted from the use of the embodied scheduling algorithm and are rejected by the automaton.

2.1.7.1. General Structure

Figure 2-3 shows the structure and the internal components of a scheduling automaton. The automaton examines each event in a history in turn. Each event is either accepted or rejected. If any individual event is rejected, then the entire history is rejected.

Each event comprises an operation, a timestamp, and a set of parameters for the designated operation. The automaton associates a precondition with each type of event operation. When considering an event,

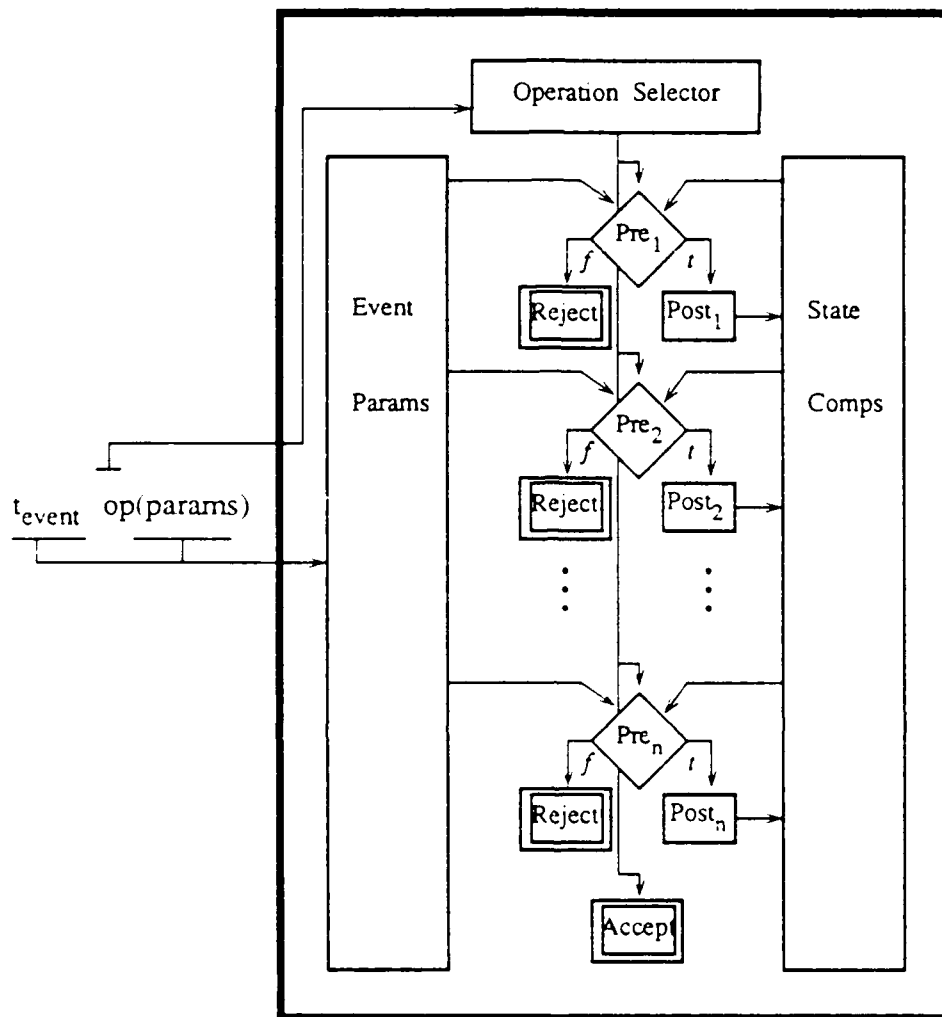


Figure 2-3: Scheduling Automaton Structure

the automaton's Operation Selector activates a test that determines whether the precondition associated with the event's operation is satisfied. If it is, then the event is accepted, and the actions specified in the postconditions for the operation are performed. If, on the other hand, the precondition for the event's operation is not satisfied, the event — and hence the entire history — is rejected.

This is illustrated in Figure 2-3. The diamond-shaped boxes represent the preconditions associated with the n event operations that may be accepted by the automaton. In a manner analogous to a flowchart, the diamond-shaped boxes have two possible outcomes, and an arrow leaves the box for each outcome. If the

precondition test fails, the arrows marked "f" indicates that the history is rejected. Otherwise, the arrow marked "t" indicates the the postconditions associated with the operation must hold.

The operation preconditions in the automaton test various conditions. These conditions may involve the values of the automaton's state components, the event timestamp, or the parameters for the event operation in question¹¹. The state components constitute the internal state of the automaton that persists across events. On the other hand, the information contained in the Event Parameters box does not persist from one event to the next — it simply represents the operation parameters and the timestamp for the current event.

The availability of this information for precondition testing is shown by the arrows leading from the State Components and Event Parameters boxes to each precondition box.

The postconditions that must hold after an event has been accepted may change some of the state component values, as indicated by the arrows leading from each postcondition box to the State Component box.

If all of the events in a history have been accepted, the Operation Selector signals the final step — shown as a single box containing the word "ACCEPT" — to declare that the history has been accepted.

2.1.7.2. Specific Scheduling Automata

The preceding discussion outlines a standard automaton framework for expressing scheduling algorithms. Each instance of a scheduling automaton for a specific scheduling algorithm would specialize this general form. This would typically involve: (1) the alteration of the preconditions and postconditions for the operations accepted by the automaton; (2) the addition of some algorithm-specific state components; and (3) the specification of a function that would select the phase to be executed at times dictated by the automaton's postconditions (or, perhaps, its preconditions).

It is largely through the last specialization — the selection function definition — that the scheduling algorithm embodied by the automaton is manifest. Different algorithms choose successor phases according to different criteria. (They may also be invoked to make selections at different times, so that the selection function alone does not completely differentiate all schedulers.)

The General Scheduling Automaton Framework is shown in Appendix A. It is a scheduling automaton that lacks a few critical pieces. While, it displays the structure of a scheduling automaton and has a number of state components, it is intentionally general and does not embody any specific scheduling algorithm. Later in this chapter (in Section 2.3.2), portions of this automaton framework will be examined in more detail.

In later chapters, specific scheduling automata of interest will be studied. These will be presented as

¹¹In principal, the originator of the event could also be tested by the precondition, but this has not proven useful to date.

extensions or specializations of the General Scheduling Automaton Framework, sharing its structure and a superset of its state components.

2.2. Assumptions and Restrictions of Model

The computational model presented is quite general. In order to focus on the questions of greatest immediate interest in this thesis, a few simplifying assumptions have been made. In particular, two specific assumptions should be stated and examined at this point.

First of all, time-value functions are restricted to be simple step functions. The most important issue to be studied in the thesis is how to use dependency information to construct a schedule that maximizes the value that an application accrues without spending too much time performing scheduling decisions. This issue is best isolated if considerations such as maximizing the value attained by completing a phase are initially ignored. This is an issue that should be dealt with in the future, but it seems like a second-order effect for most systems.

Secondly, the compute time required by an activity to complete a computational phase is assumed to be known accurately. In many real-time systems, this is a fairly reasonable assumption. Adding additional information to describe the actual distribution of computation times may increase the quality of the scheduling decisions, but it will also involve more calculations and therefore be more costly. For the simple types of computations done in typical supervisory control systems, it may well be sufficient to take the simpler approach first, at a slightly reduced cost.

2.3. Formal Model

In order to provide a precise framework in which to discuss scheduling policies for real-time activities, the following formal model has been adopted. It accommodates the aspects of the problem domain that were presented in Chapter 1 and includes all of the ideas discussed informally in the preceding sections of this chapter.

Before discussing the model itself, the notation that is employed is described, followed by definitions of key primitives in the model. Next, the formal model is presented in depth. This discussion is focused around the definition of the General Scheduling Automaton Framework. All of the other scheduling automata referred to by this work will be defined with respect to this framework. Finally, a number of observations concerning the formal model are outlined.

2.3.1. Notation and Definitions

This section describes the notation that is used throughout the rest of this and subsequent chapters. The notation is explained at this point so that all of the discussion that follows can be interpreted unambiguously.

Naming Conventions A set of conventions are employed in defining the computational model and the scheduling automata¹²:

- Identifiers written in all capital letters denote domains of values (e.g., *TIMESTAMP*, *BOOLEAN*); individual values from these domains are written in all lower-case letters (e.g., *t_a*, *true*)
- Each scheduling automaton has certain state components associated with it; these are designated by identifiers that begin with a single capital letter followed immediately by at least one lower-case letter (e.g., *Total*, *AbortClock*)
- If an automaton accepts an event in a history, the postconditions associated with the accepted event hold; when these postconditions result in modifying the value of a state component, the new value is followed by an apostrophe (e.g., *Clock' = Clock + 1* means that the new value of the automaton state component named *Clock* is one greater than the old value)

Mode-Phase Pairs. Typically, specifying the current workload of the processor is simply a matter of naming the phase that is being executed at this time. However, since it is possible to execute a phase normally or to abort it, it is necessary to refer to the computation being performed on the processor at any given time as a *mode-phase pair*. Such a pair specifies both the phase that is being executed and the mode of execution (either 'normal' or 'abort'), and it is written as an ordered pair delimited by angle brackets: $\langle m, p \rangle$.

Two auxiliary functions exist to select the individual fields from a mode-phase pair. Specifically, if $mpp = \langle m, p \rangle$, then:

$$Mode(mpp) = Mode(\langle m, p \rangle) = m$$

$$Phase(mpp) = Phase(\langle m, p \rangle) = p$$

Time-Value Functions. The simplified time-value functions studied in this work are described as step functions, where the amplitude of a function indicates the value of completing the corresponding phase on time. Let the time-value function for phase p be given by:

$$Value(p) = step(val, t_c)$$

where,

t_c is the critical time, or deadline, for this phase of an activity,

$val > 0$, is the value associated with completing a phase by its deadline,

$$step(val, t_c)(t) = \begin{cases} val, & t \leq t_c \\ 0, & t > t_c \end{cases}$$

Then define the following functions that select parameters from the simplified time-value functions:

$$Deadline(p) = DL(Value(p)) = DL(step(val, t_c)) = t_c$$

$$Val(p) = V(Value(p)) = V(step(val, t_c)) = val$$

¹²Some of these conventions and much of the notation in general has been modeled after a style used by Maurice Herlihy

2.3.2. The General Scheduling Automaton Framework (GSAF)

The General Scheduling Automaton Framework, expressed within the formal structure described in this and previous sections, provides an overall specification for the generic scheduling automaton. Although it, in fact, embodies no specific scheduling algorithm and is incompletely specified in other respects as well, the automaton framework is useful because all of the automata discussed in the rest of this work are derived by modifying it in relatively minor ways.

In the following sections, formal definitions will be given for activities, phases, shared resources, events, operations, and histories. Within this context, the various parts of the General Scheduling Automaton Framework can be expressed formally as well. These parts include the automaton state components and the preconditions and postconditions associated with the operations accepted by the automaton.

2.3.2.1. Applications and Activities

An application is composed of a set of activities, each of which comprises a sequence of computational phases. At any given time, these activities can be referred to by means of the phase that they are currently carrying out. Therefore the set of activities can be represented by the set of phases currently defined: $\{p_0, p_1, p_2, \dots\}$

2.3.2.2. Events and Histories

While executing an application, an observer located within the operating system could monitor a sequence of time-stamped events passing to and from the scheduler. These events are of the form:

where, $t_{event} \text{ op}(\text{parms}) \text{ } O$

t	is a timestamp,
op	is the operation associated with the event (as defined below),
$parms$	are the arguments for the operation,
O	is the originator of the event (either p , for a phase, or S , for the scheduler)

A sequence of these events is called a history. Notice that some of these events are generated by individual phases and some are generated by the scheduler.

2.3.2.3. Operations

The operations that may occur in events, and the potential originators of each, are shown in Figure 2-4.

The general meaning and usage of each of these operations may be stated very briefly:

- 'request-phase' — ends one computational phase and describes the requirements of the next atomically;
- 'abort-phase' — aborts the designated phase, returning all of the shared resources held by the phase to acceptable states for use by other phases;
- 'preempt-phase' — suspends the currently executing phase;
- 'resume-phase' — resumes a phase that had previously been preempted;

Operation Type	Potential Originator(s)
<i>request-phase</i> (<i>v</i> , <i>t_{expected}</i>)	Phase
<i>abort-phase</i> (<i>p</i>)	Scheduler or Phase
<i>preempt-phase</i> (<i>p</i>)	Scheduler
<i>resume-phase</i> (<i>p</i>)	Scheduler
<i>request</i> (<i>r</i>)	Phase
<i>grant</i> (<i>p</i> , <i>r</i> , <i>t_{undo}</i>)	Scheduler

where

<i>v</i>	is a time-value function,
<i>t_{expected}</i>	is the time required to execute the phase, assuming no waiting must be done to acquire shared resources.
<i>p</i>	designates a phase,
<i>r</i>	designates a shared resource, and
<i>t_{undo}</i>	is the time required to restore a shared resource to its pre-grant'ed state

Figure 2-4: Operation Types and Originators

- 'request' — signals a request for access to a shared resource;
- 'grant' — grants permission to access a shared resource.

However, the precise meaning and usage of these operations is wholly dependent upon the scheduling discipline embodied by the automaton. For instance, one automaton (embodying a FIFO or priority scheduling algorithm, for example) may deem that a new 'request-phase' event may be signaled by the currently executing activity at any time and that the activity may continue executing; while another automaton (embodying the DASA scheduling algorithm, which is presented in Chapter 3) may require that a scheduling decision must be made at that point, possibly resulting in the execution of a different activity. Similarly, the rules for when, and even if, phases may be preempted or aborted may vary from automaton to automaton.

Section 2.3.2.10 describes, in a little more detail, the semantics associated with these operations. Once again, there is some vagueness due to the fact that the definition is couched in terms of an automaton framework and not a true automaton instance. In Chapter 3, specific definitions will be presented for the operations accepted by the DASA Scheduling Automaton.

2.3.2.4. Computational Phases of Activities

The individual computational phases that comprise an activity are delimited by '*request-phase*' events. A '*request-phase*' event simultaneously ends one computational phase of an activity and describes the known requirements of the the next computational phase.

Each phase that is successfully completed contributes value to the overall application. That value is determined by evaluating the time-value function describing the phase just completed at the time of completion. On the other hand, an aborted computational phase contributes no value to the overall application — although it may free resources that allow other critical phases to execute.

2.3.2.5. Shared Resources

Phases may access shared resources. A request for such access is signalled by a phase by means of a '*request*' event for the specific resource desired. Permission to access a shared resource is signalled to the phase by means of a '*grant*' event.

All shared resources that are held by an activity must be released at the completion or abortion of each computational phase.

2.3.2.6. Phase Preemption and Resumption

At any given time there is one phase that is active. It may be preempted by the scheduler. This is signalled by a '*preempt-phase*' event. The scheduler may subsequently determine that the phase should be resumed; this is signalled by a '*resume-phase*' event.

The computational model allows a phase to be preempted at any time. Individual scheduling algorithms may restrict this by only allowing preemption at specific times or by not permitting preemption at all. This type of behavior is formally described in the precondition of the '*preempt-phase*' event operation for each specific scheduling automaton.

2.3.2.7. Event Terminology and Notation

Some additional terminology and notation will be useful for discussing events. Let an event, e represent the following event:

$$e = t_{event} \quad op(parms) \quad O$$

Then define the following simple functions:

$$timestamp(e) = t_{event}$$

$$eventtype(e) = op$$

$$source(e) = O$$

2.3.2.8. Definitions and Properties of Histories

Earlier, a history was defined as a sequence of events. Not all histories are meaningful or well-formed. Let $e_0, e_1, e_2, \dots$ denote events. Then, formally, a history, H , can be denoted as:

$$H = e_0 \cdot e_1 \cdot e_2 \cdot \dots \cdot e_n$$

where the operator " $\cdot$ " denotes concatenation.

Informally, a projection of a history selects certain events from a history, preserving their relative positions in the projection. Therefore, a projection of a history could include all of the 'request-phase's from the history or all of the events that dealt with a specific phase. The symbol " $|$ " denotes a projection. So for example, $H|p$ represents the projection of history H onto phase p . This projection would include all of the events that were originated by phase p or that were originated by the scheduler and included p as an operational parameter.

The conditions that define a well-formed history include¹³:

- event timestamps must increase monotonically and must be unique — *test: examine the timestamps on events; for example, apply the function timestampsOK() to a history H to verify that it meets this requirement, where timestampsOK() is defined as:*

$$\text{timestampsOK}(\phi) = \text{timestampsOK}(e) = \text{true}$$

$$\text{timestampsOK}(e_1 \cdot e_2 \cdot H) = \begin{cases} \text{false,} & \text{if } \text{timestamp}(e_1) \geq \text{timestamp}(e_2) \\ \text{timestampsOK}(H), & \text{otherwise} \end{cases}$$

- request for a resource must appear in the schedule before the corresponding grant — *test: for each 'grant' event, search the history of the phase in which the 'grant' occurred for a preceding 'request' for the same resource*
- a phase cannot be preempted if it is not active; it cannot be resumed if it is active; and so on — *simple tests check all of these conditions*
- a given phase either commits or aborts; the events assure that a single phase cannot do both; however, a well-formed history must have at most one 'abort-phase' event for any given phase — *test: examine the history for the occurrence of two or more 'abort-phase' events for a single activity that are not separated by a 'request-phase' event.*
- expected compute time is accurate — *test: check that the estimated computation time equals the actual computational time used; for example, the following test could be applied:*

$$\text{ctest}(H) = (\forall p)(\text{comptimeOK}(H|p) \vee \text{phaseaborted}(H|p) \vee \text{phaseunfinished}(H|p))$$

where,

¹³It is not always clear that a specific test be a requirement of a well-formed history or whether it is a requirement that determines which histories will be accepted by a given automaton. There is no question that the proper temporal ordering of events is a requirement for a well-formed history; however, tests that constrain the relative ordering of specific events — for instance, 'request' and 'grant' events — in a history are not so obviously requirements for a well-formed history. As a result, this list is merely an attempt to lay down an initial set of tests. Some of these tests need not be done prior to submitting the history to an automaton — in those cases, the automaton will enforce the requirements verified by the tests in question.

$$comptimeOK(p, \emptyset) = comptimeOK(p, e) = 0$$

$$comptimeOK(p, e_1 \cdot e_2 \cdot H) =$$

$t_2 - t_1 + comptimeOK(H),$	$if(e_1 = t_1 \text{ resume-phase}(p) \ S$ $\vee e_1 = t_1 \text{ grant}(p) \ S)$ $\wedge (e_2 = t_2 \text{ preempt-phase}(p) \ S$ $\vee e_2 = t_2 \text{ request}(r) \ p)$
$t_2 - t_1,$	$if(e_1 = t_1 \text{ resume-phase}(p) \ S$ $\vee e_1 = t_1 \text{ grant}(p) \ S)$ $\wedge (e_2 = t_2 \text{ request-phase}(v, t) \ p$ $\vee e_2 = t_2 \text{ abort-phase}(p) \ O)$

$$phaseaborted(p, \emptyset) = false$$

$$phaseaborted(p, e \cdot H) =$$

$true,$	$if \ e = t \ \text{abort-phase}(p) \ O$
$false,$	$if \ e = t_1 \ \text{request-phase}(v, t_2) \ p$
$phaseaborted(p, H),$	$otherwise$

$$phaseunfinished(p, \emptyset) = true$$

$$phaseunfinished(p, e \cdot H) =$$

$false,$	$if \ e = t \ \text{abort-phase}(p) \ O$
$phaseunfinished(p, H),$	$\vee \ e = t_1 \ \text{request-phase}(v, t_2) \ p$
	$otherwise$

- expected abort time is accurate — *test: similar to the previous test*
- estimated computation time required for a phase must always be greater than or equal to zero¹⁴
— *test: straightforward inspection of each 'request-phase' event in the history*
- no 'request' event should request shared access to the nullresource — *test: straightforward inspection of each 'request' event in the history*

2.3.2.9. Automaton State Components

The state components associated with the General Scheduling Automaton Framework are shown in Figure 2-5. Each component and the range of values it may take on, is described below.

ExecMode. *ExecMode* is a relation that associates an execution mode with each phase. At any given time, a phase can be either executing normally or aborting. Also at any time, a normally executing phase can be aborted. Once an abort is initiated, it must be completed before normal execution of the entire phase can again be attempted.

ExecClock and AbortClock. The next two state components shown in Figure 2-5 are used to track the amount of time required to complete the normal execution or the abortion of a phase. When a phase is executing normally, the relation *ExecClock* indicates the amount of processing time needed to successfully

¹⁴An additional requirement may also be placed on the parameters of a 'request-phase' event: the value function must be of the appropriate form, as outlined below. This requirement has not been included in this list because the tests that are present all apply to the general case of scheduling with dependency considerations in a real-time environment using information available from arbitrary time-value functions. This requirement is related to a simplification made to make the work more clear and more manageable, and so does not seem to carry the same weight as the others listed above.

General State Components:

- ExecMode: PHASE $\rightarrow$ MODE (MODE is either 'normal' or 'abort')
- ExecClock: PHASE $\rightarrow$ VIRTUAL-TIME
- AbortClock: PHASE $\rightarrow$ VIRTUAL-TIME
- ResumeTime: PHASE $\rightarrow$ TIMESTAMP
- Value: PHASE $\rightarrow$ (TIMESTAMP $\rightarrow$ VALUE)
- Total: VALUE (initially '0')
- RunningPhase: PHASE (initially 'nullphase')
- PhaseElect: MODE $\times$ ¹⁵ PHASE (initially '<normal, nullphase>')
- PhaseList: list of PHASE (initially '0')

Domains for State Component Values:

- MODE: normal $\vee$ abort
- PHASE: $\in \{p_0, p_1, p_2, \dots\} \vee$ nullphase
- RESOURCE: $\in \{r_0, r_1, r_2, \dots\} \vee$ nullresource
- TIMESTAMP: real number, expressed in ticks of standard clock
- VALUE: real number ≥ 0
- VIRTUAL-TIME: real number ≥ 0 , expressed in ticks of standard clock

Figure 2-5: State Components of General Scheduling Automaton Framework

complete the execution of that phase. Similarly, when a phase is aborting, *AbortClock* indicates the amount of processing time needed to complete the abort processing.

At the start of a new phase, *ExecClock* associates a value provided by the activity with the phase. If the phase was executed in isolation¹⁶, *ExecClock* specifies the amount of time that would elapse before the phase would complete executing. Each time the phase executes, the value of *ExecClock* for that phase decreases. When it reaches zero, then the phase has completed execution.

On the other hand, *AbortClock* represents the time required to abort the current phase. In addition, the exact length of time required to abort the phase depends on the number and type of shared resources that it has acquired. Therefore, since no shared resources have yet been acquired, *AbortClock* is zero at the start of every phase. Subsequently, after any shared resource is requested and granted, the value of *AbortClock*

¹⁵This designates a cross product. That is, *PhaseElect* is actually a mode-phase pair, as described in Section 2.3.1.

¹⁶By executing in isolation, contention with other activities for both processor cycles and shared resources is eliminated. In fact, the phase does not execute in isolation and these factors cannot be ignored — leading to this scheduling work.

is incremented by an amount that is a function of that resource. This amount of time has been chosen to allow the shared resource to be returned to an acceptable state so that other phases may use it.

ResumeTime. *ResumeTime* associates with each phase the time at which it last resumed execution. This value is useful in keeping the values of *ExecClock* and *AbortClock* accurate for the executing phase. Whenever the currently executing phase is surrendering the processor, *ResumeTime* can be compared to the current time to determine the amount of computation time consumed by the phase — thus allowing *ExecClock* or *AbortClock* to be updated, depending on the current execution mode.

Value. The relation *Value* associates time-value functions with phases. In this case, the time-value functions are themselves represented by relations: given a time, a time-value relation will return the value accrued by completing the phase at that time. As stated in Section 2.2, the time-value functions considered in this work are simple step functions.

Total. *Total* accumulates the values accrued by successfully completing phases. Initially, since nothing has been accomplished, *Total* is zero. Then, after any phase is successfully completed, the amount of value indicated by the phase's *Value* relation for that completion time is added to *Total*.

The values of the *Total* state components of two different scheduling automata that have worked on the same application can be compared to determine which yielded a higher total value for the application. (This fact will be used in simulations and proofs in later chapters when comparing two different scheduling algorithms.)

RunningPhase. *RunningPhase* indicates which phase is currently executing on the processor. If no activity is currently executing — as is the case initially — *RunningPhase* is equal to *nullphase*.

PhaseElect. *PhaseElect* also indicates a phase. In this case, it is the phase that should be executing now. If this is different than *RunningPhase*, then the currently executing phase should be suspended and replaced by the *PhaseElect*. Once again, in the initially empty system, *PhaseElect* specifies that the *nullphase* should be executed normally.

PhaseElect names not only the phase to be executed but also the execution mode for the phase.

PhaseList. *PhaseList* is simply a list that containing all of the phases known to the automaton. This list changes as new phases arrive and old phases are completed. Initially, since there are no phases *PhaseList* is empty.

Automaton-Specific State Components. Other state components are also associated with an automaton. These are used to handle some of the bookkeeping details for the specific scheduler being used. The components that appear above are intended to reflect the state that any specific scheduler would need and maintain under this general model.

Specific initial values may be given to many of these state components in order to satisfy the requirements of a given automaton.

Domains for State Component Values. The domains that supply the values for the state components are straightforward and are shown in Figure 2-5 along with the state components of the General Scheduling Automaton Framework. The domain *MODE* contains only two values: *normal* and *abort*. The domain *PHASE* consists of all of the phases known by the automaton as well as the *nullphase*. Similarly, the domain *RESOURCE* consists of all of the shared resources known to the automaton as well as the *nullresource*. The values from all of the time-value functions are drawn from the domain *VALUE*. These must be positive (according to the assumptions stated earlier in Section 2.3.1) and are chosen from the real numbers so that there are no unnecessary restrictions placed on them. The domain *VALUE* also contains zero since *Total* receives its value from this domain and it initially has no accrued value.

Time is central to the behavior of real-time systems, and the domain *TIMESTAMPS* provides a source of markers in time for the automaton to use. Each timestamp is expressed in terms of ticks of a standard clock. The ticks of this clock are equally spaced in time; and in fact, nothing in the model prevents the timestamps from taking on fractional numbers of ticks — thus allowing arbitrarily great precision to be obtained in representation of times.

The other domain related to time is the *VIRTUAL-TIME* domain. Values from this domain represent time durations. Once again, they are expressed in terms of ticks of the standard clock. These durations are used to supply values for state components like *ExecClock* and *AbortClock* where only non-negative durations are meaningful.

2.3.2.10. Operations Accepted by GSAF with Preconditions and Postconditions

The operations recognized by the General Scheduling Automaton Framework are shown in Figure 2-6. (Notice that this figure has two parts, appearing on pages C-34 and C-35, respectively.) Minimal, or skeletal, preconditions and postconditions for each operation are included in the figure.

In later chapters, some specific scheduling automata will be discussed in detail. Each discussion will include a description of the preconditions and postconditions associated with the operations accepted by the automaton under consideration. Consequently, the discussion of those topics in this section will be brief. Only the highlights and general structure of an automaton's operation specification will be addressed here.

Since the 'request-phase' event denotes the initiation of each computational phase of every application activity, it is accepted by every scheduling automaton. Furthermore, its precondition is simply "*true*", indicating that new phases can arrive at any time. This does not necessarily require that a new scheduling decision must be made upon the arrival of each new phase, although some automata may do just that. Since such a scheduling decision is not made in every scheduling automaton, no decision is made in the General Scheduling Automata Framework.

In the same spirit, the postconditions for the 'request-phase' event include only those conditions that will almost certainly belong in every scheduling automaton of interest. Those postconditions: (1) accumulate any value accrued from completing a previous computational phase of the same activity; (2) initialize the automaton's state components to capture the new phase's scheduling parameters; and (3) update the list of phases known to the automaton based on the new phase's scheduling parameters.

```

•  $t_{event}$  request-phase( $v, t_{expected}$ )  $p$ :
  preconditions:
    true <No preconditions here so that interrupts and other new phases
      can occur at any time>
  postconditions:
    if (RunningPhase =  $p$ ) then
      if (ExecMode( $p$ ) = normal) then
        Total' = Total + Value( $p$ )( $t_{event}$ )
      else
        ;no value for aborted phase
        ;release the resources acquired during the phase

    ;accept values for scheduling parameters
    Value'( $p$ ) =  $v$ 
    ExecClock'( $p$ ) =  $t_{expected}$ 
    AbortClock'( $p$ ) = 0
    ExecMode'( $p$ ) = normal
    ;note that  $p$  is not resource-waiting

    ;make sure  $p$  is part of the list of phases, if necessary
    if ( $t_{expected} > 0$ ) then
      PhaseList' = PhaseList  $\cup$  { $p$ }
    else
      PhaseList' = PhaseList - { $p$ }

•  $t_{event}$  abort-phase( $p$ )  $O$ :
  preconditions:
    <Specific to the scheduler under consideration>
  postconditions:
    ExecMode'( $p$ ) = abort
    ResumeTime'( $p$ ) =  $t_{event}$ 

```

Figure 2-6: Operations Accepted by General Scheduling Automaton

Notice that value is accumulated for the completion of a previous phase only if the currently executing phase issues the 'request-phase' event, thereby signaling that the current phase has completed execution. If some other activity issues the 'request-phase' event, it is signaling the existence of a new phase to the automaton while a different phase is executing, so no phase completion has occurred.

In addition, no value is accrued for a phase that has been aborted. If, on the other hand, the phase has completed successfully the value accrued is determined by evaluating its time-value function at the time of completion.

Finally, since it requires a positive amount of time to accomplish any processing, a $t_{expected}$ parameter that is less than or equal to zero indicates that there is no subsequent computational phase for the activity

-
- t_{event} *preempt-phase(p) S*:
 - preconditions:
 $\langle \text{Specific to the scheduler under consideration} \rangle$
 - postconditions:
 if (ExecMode(p) = normal) then
 $\text{ExecClock}'(p) = \text{ExecClock}(p) - (t_{event} - \text{ResumeTime}(p))$
 else
 $\text{AbortClock}'(p) = \text{AbortClock}(p) - (t_{event} - \text{ResumeTime}(p))$
 - t_{event} *resume-phase(p) S*:
 - preconditions:
 $\langle \text{Specific to the scheduler under consideration} \rangle$
 - postconditions:
 $\text{ResumeTime}'(p) = t_{event}$
 - t_{event} *request(r) p*:
 - preconditions:
 $\langle \text{Specific to the scheduler under consideration} \rangle$
 - postconditions:
 $\text{ExecClock}'(p) = \text{ExecClock}(p) - (t_{event} - \text{ResumeTime}(p))$
 - t_{event} *grant(p, r, undotime(r)) S*:
 - preconditions:
 $\langle \text{Specific to the scheduler under consideration} \rangle$
 - postconditions:
 $\text{ResumeTime}'(p) = t_{event}$
 $\text{AbortClock}'(p) = \text{AbortClock}(p) + \text{undotime}(r)$ ¹⁷
-

Figure 2-6: Operations Accepted by General Scheduling Automaton, *continued*

issuing the 'request-phase' event. In that case, the phase is removed from the *PhaseList*; in all other cases, the phase is included in the *PhaseList*.

The 'abort-phase' event in the General Scheduling Automaton Framework is similar to the remainder of the scheduling events: its precondition is automaton-specific and its postconditions specify bookkeeping that must be done in the event that the event occurs. In particular, the 'abort-phase' event's postconditions change the phase's execution mode to *abort* and note the time that the phase began aborting. In the event of a preemption, this time (*ResumeTime*) will be consulted to adjust the phase's *AbortClock* to indicate the amount of time required to complete the abort processing, which will be used in subsequent scheduling decisions.

¹⁷The function 'undotime()' indicates the amount of time that will be required to restore the resource just acquired to an acceptable state for use by another activity. In many cases, this may simply involve returning the resource to the state it had at the time it was acquired. In other cases, returning the resource to any of a number of semantically equivalent states may be sufficient or actions may have to be performed to affect the physical process under controlled. The actions required and the amount of time they will take may vary from system to system and from application to application. Consequently, for the purposes of this work, they have been cast as a function that acts to indicate their role without applying a single definition across all resources or applications.

The 'preempt-phase' event has an automaton-specific precondition. Its postconditions handle the bookkeeping associated with preempting the executing phase. Specifically, *ExecClock* or *AbortClock* is updated to reflect the amount of time still required to complete the normal or abort processing of the phase, respectively. This is accomplished by subtracting the amount of time the phase had executed prior to the preemption from the amount of time it still needed to complete processing before it began that execution.

A 'resume-phase' event is used to resume execution of a phase that had been suspended by a 'preempt-phase' event. The 'resume-phase' event, which has an automaton-specific precondition, simply notes the time at which the designated phase resumed execution. This time is used to adjust the state components dealing with the required execution time of the phase whenever the phase is subsequently preempted.

Once again, the 'request' event has an automaton-specific precondition. Its postcondition updates the appropriate state component clock for the phase, depending on its execution mode. This is done to facilitate the use of a scheduling decision as a result of a request for a shared resource. The updating of the relevant state components ensures that the automaton will make a decision based on the most up-to-date information.

The 'grant' event, which also has an automaton-specific precondition, notes the time at which the phase is awarded the shared resource it had previously requested and begins execution. Another postcondition increments the *AbortClock* state component for the designated phase to reflect the amount of time that will be required to return the shared resource to an acceptable state for another phase in the event that the current phase is aborted. This length of time may vary from resource to resource, and so is denoted as *undotime(r)*, a function of the resource in question.

Although the 'request' and 'grant' phase events behave as if the processor is surrendered after each request, this does not have to be the case. The 'request' event can be immediately followed by the corresponding 'grant' event to model the situation in which the processor is not surrendered.

Typeface Convention. In the definition of the GSAF, all of the operation definitions — their preconditions and postconditions — have been presented with a roman (normal) typeface. In the future, when automata are presented, those parts that are common with the GSAF will continue to be written in a roman typeface. However, those parts that are different will be written in an italic typeface. Hopefully, this will allow the reader to focus on those parts of the definition that are different from the general framework.

2.3.2.11. Active Phase Selection

Although the General Scheduling Automaton Framework contains state components and will accept some scheduling events, it is not really a scheduling automaton. Rather, it is a framework: a skeleton that has most, but not all, of the elements of a scheduling automaton. For instance, as was discussed in the previous section (Section 2.3.2.10), most of the preconditions for accepting various scheduling events are unspecified in the General Scheduling Automaton Framework. Also, while some postconditions have been specified, they have not been completely specified.

Another of the more noticeable omissions in the specification of the General Scheduling Automaton Framework is the lack of a function to select the next phase to execute. Furthermore, not only is this function not specified, the places in the automaton where it is to be invoked are also unspecified. This is because different schedulers invoke this function at different times. Therefore, there is no canonical set of times (corresponding to a fixed set of points in the General Scheduling Automaton Framework) where all scheduling algorithms invoke a phase selection function. As a result, this has been omitted from the General Scheduling Automaton Framework, which acts as a lowest common denominator of sorts among instance of scheduling automata.

To illustrate this point, consider two simple scheduling algorithms: FIFO scheduling and priority scheduling. Whenever a new computational phase enters the system — as indicated by a new 'request-phase' event — the FIFO scheduling automaton will note that fact, but will not invoke a phase selection function to determine what phase to execute. It simply allows the currently executing phase to proceed until it gives up the processor.

On the other hand, the priority scheduling automaton will make a new determination concerning which phase to execute: if the new computational phase has a higher priority than the currently executing phase, then the active phase is preempted in favor of the new phase.

More complex scheduling algorithms may evaluate the phase selection function at other times as well. For instance, the DASA algorithm described in Chapter 3 invokes the phase selection function whenever a shared resource is requested by any phase. It is also conceivable that there are schedulers that might select which phase to run asynchronously with respect to the given set of scheduler operations. For example, a round robin scheduler that gave each phase a turn by offering it a time-slice would make preemption decisions following evenly spaced clock interrupts. To accommodate such extensions, new scheduling operations would have to be added to the model. While that is straightforward, it is not required to investigate the algorithms of interest for scheduling supervisory control systems, and so it would only serve to complicate the model. Consequently, the scheduling operations included in the model represent a minimal set that captures all of the relevant behavior within supervisory control systems.

2.3.3. Notes

A few fairly minor facts can be noted concerning the GSAF framework. The following paragraphs address some of them. Most of them explain how facets of real applications are, or can be, reflected in the formal model.

2.3.3.1. Manifestation of Assumptions and Restrictions

Section 2.2 describes the specific assumptions and restrictions that are employed in this work. A look at the GSAF framework will reveal how those assumptions are manifest.

The first assumption stated that all time-value functions are restricted to be simple step functions. The definition provided for time-value functions in the model in Section 2.3.1 captures that assumption directly.

are often not explicitly recognized as scheduling events. Specifically, the resource-related events — 'request' and 'grant' — are directly presented to the scheduler because they may well result in new scheduling decisions.

This should be contrasted with many other models and operational systems. There, there are two separate operating system facilities: a scheduler and a resource manager. (See Figure 2-7.) The resource manager may be representing one or more actual system managers — the lock manager and the semaphore manager, for instance.

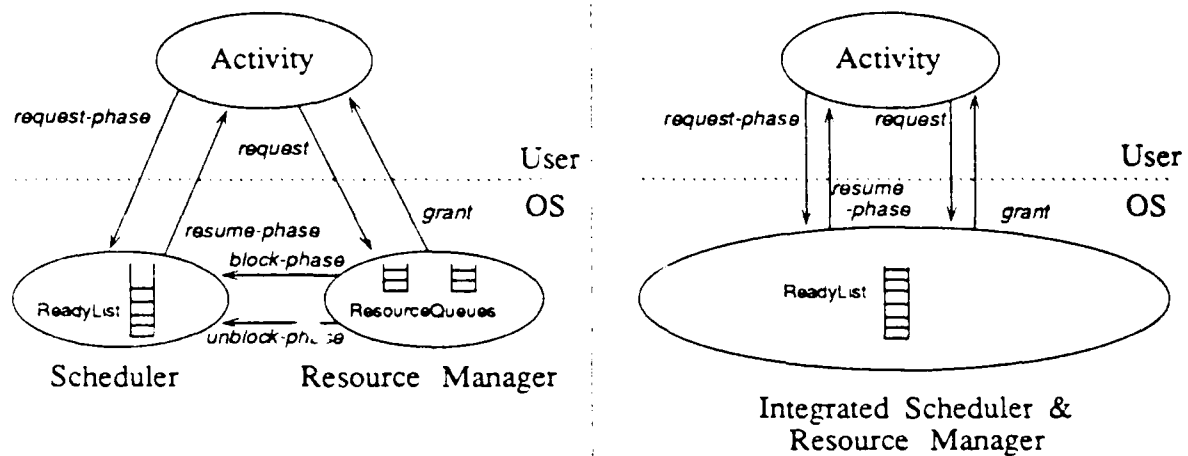


Figure 2-7: Organizations of Scheduling Functions

The difference in organization is often significant. When there are separate managers handling access to resources, they will often make implicit scheduling decisions that are not in keeping with the overall goals of a real-time system. For example, resource managers may service resource requests in a first-come-first-serve manner. In that case, a request may be placed in a FIFO (first in, first out) queue if the desired resource is not currently available. As a result, not only is the requesting activity blocked when it is enqueued, it is not even considered by the scheduler again until it has been removed from the queue. In effect, all of the activities that preceded it in the queue were given precedence over it, regardless of the relative urgency of their time constraints or any other dependency considerations.

It would be much more appealing to apply the same type of algorithm to select which activity in the queue should receive access to the resource next that is used in selecting which activity should execute next in general. The work done here does employ such an integrated approach to scheduling, and the interfaces described in the model reinforce this integrated scheduling notion.

Chapter 3

The DASA Algorithm

The primary algorithm investigated in this thesis is called the DASA (Dependent Activity Scheduling Algorithm) algorithm. It addresses the dependency concerns described in the previous chapters in a clear and natural manner. The algorithm is based on a set of heuristics that deliver the type of behavior required in real-time systems.

This chapter presents the DASA algorithm. First, the algorithm is described in general terms, placing emphasis on the rationale for the algorithm. Then a formal definition is discussed, providing a framework for careful analysis of the algorithm. Finally, the scheduling example from Section 1.3 is revisited. This time the DASA algorithm is employed to make all of the scheduling decisions to contrast its behavior with that of the algorithms previously used.

3.1. Dependent Activity Scheduling Algorithm

In this section, the underlying heuristics for the DASA algorithm will be described, along with the rationale for their adoption. This should help to explain the high level goals for the algorithm. That discussion will be followed by an informal definition of the DASA algorithm. (The next section, Section 3.2, will provide the formal definition.)

3.1.1. Rationale for Heuristics

The DASA algorithm was constructed to possess a set of properties, each of which is logical and has appeal on its own merits. Taken together, they suggest that the algorithm will be quite effective in handling scheduling problems with dependency considerations.

Before looking at the definition of the DASA algorithm, two important metrics must be understood. These are the notions of *value density* [Locke 86] and *potential value density*. Value density is a measure of how much value (as defined by the application) per unit time will be acquired by executing a phase. In the cases considered by this thesis, where time-value functions are simply step functions, the value density is the size of the step function — the value — by the required computation time¹⁹.

¹⁹In more complex cases, more involved time-value functions and less certain computation time requirements are considered.

Potential value density extends the notion of value density to a collection of phases composed of a designated phase and a set of phases on which it depends. In fact, the potential value density of such a collection of phases is the total of their individual values divided by the total required computation time for all of the phases in the collection. Furthermore, since phases may be aborted at any time, aborting phases must be handled differently than those that are executing normally — an aborting phase contributes no value, but it does require computation time. Therefore, aborting a computation will always act to reduce the potential value density of a collection of phases. The advantage that aborts offer is that they may greatly reduce the delay that must be incurred before starting the execution of a designated computation.

With these metrics in mind, the properties desired for the DASA algorithm can be reviewed:

1. explicitly account for dependencies — account (in terms of both time required and in potential value available) for not only a phase, but also for other phases on which it depends;
2. minimize effort — apply the minimum amount of effort necessary to allow a phase to be scheduled (use aborts to expedite this process);
3. maximize return/benefit — examine phases in order of decreasing potential value density, thereby always obtaining the greatest return (in value) on a given investment (of time);
4. maximize the chance of meeting deadlines — approximate a deadline scheduler insofar as possible;
5. globally optimize schedule — review the schedule constructed incrementally and collapse/remove redundant or unnecessary steps.

3.1.2. The DASA Algorithm

Since mutual dependencies among activities may arise during the course of execution, the DASA algorithm actually has two major parts: the Dependency Scheduling Algorithm, which, given a set of phases (and their scheduling parameters) without circular dependencies, will select the next phase to be run, and the Deadlock Resolution Algorithm, which performs a similar function when there are circular dependencies.

The following two sections describe each of these component algorithms in detail.

3.1.3. The DASA Algorithm: Dependency Scheduling

The DASA algorithm conforms to the computational model defined in Chapter 2 and meets the problem requirements, while also possessing the properties listed in Section 3.1.1 above.

The following fragment illustrates how the potential value density (PVD) for a phase p is calculated for use in DASA:

$$\begin{aligned}
 PVD(p) &= \begin{cases} 0, & \text{if } p \text{ is aborting} \\ \frac{Val(p) + PV(Dep(p))}{ExecClock(p) + PT(Dep(p))}, & \text{otherwise} \end{cases} \\
 PV(p) &= \begin{cases} 0, & \text{if } p = \text{nullphase} \\ 0, & \text{if quicker to abort } p \text{ than to complete } p \\ Val(p) + PV(Dep(p)), & \text{otherwise} \end{cases} \\
 PT(p) &= \begin{cases} 0, & \text{if } p = \text{nullphase} \\ \langle \text{time to abort } p \rangle, & \text{if quicker to abort } p \text{ than to complete } p \\ \langle \text{time to complete } p \rangle + PT(Dep(p)), & \text{otherwise} \end{cases} \\
 Dep(p) &= \begin{cases} \text{nullphase}, & \text{if } p \text{ is ready to run} \\ \langle \text{phase on which } p \text{ depends} \rangle, & \text{otherwise} \end{cases}
 \end{aligned}$$

Notice that this calculation demonstrates a property mentioned earlier: the least amount of time possible is expended to make the phase ready to run. That is why the decision is made to abort if that will result in a shorter delay before the phase in question is ready to execute.

For any phase, the set of phases on which it depends, either directly or indirectly, and which must therefore be completed or aborted before it can run is called its *dependency list*. In the definition for PVD, the set of phases examined while evaluating $Dep(p)$ constitutes the dependency list for a phase. The general concept of a dependency list could also be used by other algorithms similar to DASA, although their specific definition of the dependency list might vary somewhat to reflect a different set of desired properties.

A simplified procedural version of the DASA Dependency Scheduling Algorithm is shown in Figure 3-1.

-
1. create an empty schedule
 2. determine dependency list and PVD for each phase
 3. if deadlock is detected, use DASA Deadlock Resolution Algorithm
 4. sort phases according to PVD
 5. examine each phase in turn (highest PVD first)
 - a. tentatively add phase and dependencies to schedule
 - b. test feasibility of schedule
 - c. if feasible, make tentative changes; else, discard them
 - d. reduce schedule, if possible
-

Figure 3-1: Simplified Procedural Definition of DASA Scheduling Algorithm

Notice that this scheduling algorithm considers all of the existing phases each time a scheduling decision is made. Most scheduling algorithms do not do this — they typically consider only those that are ready to

run immediately, not phases that are blocked awaiting access to shared resources. A critical objective of the DASA algorithm is to take advantage of this additional information to improve the quality of scheduling decisions. It is information that the system could always examine; but in non-real-time systems, there is no motivation to look at it.

3.1.4. The DASA Algorithm: Deadlock Resolution

The work that has been done up to this point focuses on the Dependency Scheduling Algorithm portion of the problem. The Deadlock Resolution Algorithm must still be devised, although it can be anticipated that it will have properties and use methods that are similar to those employed by the Dependency Scheduling Algorithm.

3.2. Formal Definition of DASA

Thus far, the rationale and informal description of the DASA algorithm have been presented. In order to provide a rigorous specification that will permit analytic study of the algorithm, a more formal definition is required. That definition is presented in the following section along with explanations of interesting and important points.

3.2.1. The Formal Definition

The formal definition is cast in terms of the automaton model presented in Chapter 2. Remember that a scheduling automaton examines histories of scheduling events and either accepts or rejects them. A history is accepted by a scheduling automaton if and only if the sequence of events comprising the history could have been generated by the scheduling algorithm embedded within the automaton.

Although all scheduling automata share a common framework, each individual automaton has several unique parts: (1) its state components; (2) the scheduling events that it recognizes — including the preconditions and postconditions associated with recognizing each event and the changes that occur in state component values as a result; and, of course, (3) the scheduling algorithm that is embedded within the automaton. Each of these parts is formally defined in the sections that follow for the DASA Scheduling Automaton.

3.2.1.1. DASA Automaton State Components

The DASA algorithm considers all existing phases each time a scheduling decision is made. In the formal definition that follows, let the set of phases currently known to the automaton be represented as $\{p_0, p_1, p_2, \dots\}$.

Similarly, let the set of resources currently known to the automaton be represented as $\{r_0, r_1, r_2, \dots\}$.

The state components associated with the DASA scheduling automaton are presented in Figure 3-2.

General State Components:

- ExecMode: PHASE $\rightarrow$ MODE (MODE is either 'normal' or 'abort')
- ExecClock: PHASE $\rightarrow$ VIRTUAL-TIME
- AbortClock: PHASE $\rightarrow$ VIRTUAL-TIME
- ResumeTime: PHASE $\rightarrow$ TIMESTAMP
- Value: PHASE $\rightarrow$ (TIMESTAMP $\rightarrow$ VALUE)
- Total: VALUE (initially '0')
- RunningPhase: PHASE (initially 'nullphase')
- PhaseElect: MODE $\times$ PHASE (initially '<normal, nullphase>')
- PhaseList: list of PHASE (initially 'ø')

Algorithm-Specific State Components:

- Owner: RESOURCE $\rightarrow$ PHASE (initially 'nullphase' for each resource)
- ResourcesHeld: PHASE $\rightarrow$ list of RESOURCE
- ResourceRequested: PHASE $\rightarrow$ RESOURCE (initially 'nullresource'; also note: ResourceRequested(nullphase) = 'nullresource')

Domains for Value Types:

- MODE: normal $\vee$ abort
- PHASE: $\in \{p_0, p_1, p_2, \dots\} \vee$ nullphase
- RESOURCE: $\in \{r_0, r_1, r_2, \dots\} \vee$ nullresource
- TIMESTAMP: time, expressed in ticks of standard clock
- VALUE: real number ≥ 0
- VIRTUAL-TIME: real number ≥ 0 (represents a time duration)

Figure 3-2: State Components of DASA Scheduling Automaton

There are two distinct groups of state components shown: general state components, which are found in any scheduling automaton, and algorithm-specific state components, which are defined only for a particular scheduling automaton.

The general state components were discussed in Chapter 2. They include a number of components that describe important characteristics of each individual phase (*ExecMode*, *ExecClock*, *AbortClock*, *ResumeTime*, and *Value*), as well as components that indicate the status of the automaton itself (*Total*, *RunningPhase*, *PhaseElect*, and *PhaseList*).

All of the algorithm-specific state components of the DASA Scheduling Automaton deal with requesting

and holding shared resources. The relation *Owner* indicates which, if any, phase currently possesses each of the shared resources. The *Owner* of all unassigned resources is *nullphase*. The *ResourcesHeld* relation associates with each phase the list of resources that have been granted to that phase. And finally, the *ResourceRequested* relation specifies which resource a given phase desires. Whenever, there is no unsatisfied resource request for a phase, the corresponding *ResourceRequested* value is *nullresource*.

The bottom portion of Figure 3-2 defines the values that each of the state components may assume. All of these are general value domains that were discussed when the scheduling automaton model was presented in Chapter 2. They are repeated here only for convenience — they allow the relation definitions to appear in context so that earlier material need not be consulted.

An initial value is shown for many of the state components. These values indicate that, at the outset, there are no phases known to the automaton, no value has been accrued, all of the shared resources are available, and the processor is idle. Each of the relations that provide information for each phase in the system is initially empty since there are no phases. As phases arrive (indicated by issuing *request-phase* events), entries are made in each of these relations.

Definitions. or anywhere else]

Each activity/phase has a state associated with it. It is either *running* or it is *blocked*. If it is blocked, it may have been *preempted* or it may have blocked to *wait* on a resource that was unavailable when requested.

$$Running(p) \equiv p = RunningPhase$$

$$Blocked(p) \equiv p \neq RunningPhase$$

$$ResourceWaiting(p) \equiv (\exists r)(ResourceRequested(p) = r \wedge r \neq nullresource \wedge Owner(r) \neq p)$$

$$Preempted(p) \equiv Blocked(p) \wedge \neg ResourceWaiting(p)$$

Access Queues for Resources. There is one state component that is not present in the DASA Scheduling Automaton but is commonly found in other scheduling automata for this problem domain: a relation that, given a resource, specifies the queue of phases that are waiting for access to the resource. That state component is not found in this automaton because it tends to reflect an ordering among pending requests for a shared resource — for example, requesters may be served in a FIFO fashion or according to their priority. While the DASA algorithm will in some sense order such requests, it is done in a completely dynamic fashion. The needs of each phase, including access to shared resources, are considered along with the benefit of executing the phase each time a scheduling decision is made.

3.2.1.2. Operations Accepted by DASA Automaton

The operations recognized by the DASA scheduling automaton and their preconditions and postconditions are shown in Figures 3-3, 3-4, and 3-5. Figure 3-3 presents the 'request-phase' operation, which is used to initiate each computational phase of the activities comprising the application. Figure 3-4

depicts the other operations involving phases that are recognized by the DASA Scheduling Automaton. And Figure 3-5 shows those operations that deal specifically with shared resources. The following paragraphs describe each of these operations in detail.

```

•  $t_{event}$  request-phase( $v, t_{expected}$ )  $p$ :
  preconditions:
    true <No preconditions here so that interrupts and other new phases
        can occur at any time>
  postconditions:
    if (RunningPhase =  $p$ ) then
      if (ExecMode( $p$ ) = normal) then
        Total' = Total + Value( $p$ )( $t_{event}$ )
      else
        ;no value for aborted phase
        ;release the resources acquired during the phase
        for  $r$  in ResourcesHeld( $p$ )
          Owner'( $r$ ) =  $\emptyset$ 
        ResourcesHeld'( $p$ ) =  $\emptyset$ 
    Value'( $p$ ) =  $v$ 
    ExecClock'( $p$ ) =  $t_{expected}$ 
    AbortClock'( $p$ ) = 0
    ExecMode'( $p$ ) = normal
    ;note that  $p$  is not resource-waiting

    ;make sure  $p$  is part of the list of phases, if necessary
    if ( $t_{expected} > 0$ ) then
      PhaseList' = PhaseList  $\cup$  { $p$ }
    else
      PhaseList' = PhaseList - { $p$ }
    PhaseElect' = SelectPhase(PhaseList'20)
    if ( $p$  = RunningPhase) then
      ;give up processor until next 'resume-phase'
      RunningPhase = nullphase
    else
      ;happened under interrupt—leave 'RunningPhase' alone

```

Figure 3-3: 'RequestPhase' Operation Accepted by DASA Scheduling Automaton

Request-Phase. The 'request-phase' operation delimits computational phases for an activity. Each

²⁰Here, the value assigned to one state component (PhaseList') in these postconditions is used to determine the value of another state component (PhaseElect') in the same group of postconditions. In the interests of convenience and clarity, this has been done, rather than writing all of the new state component values in terms of only the old state component values. Of course, it is possible to express the new value of PhaseElect in terms of the old state component values, as follows:

```

if ( $t_{expected} > 0$ ) then
  PhaseElect' = SelectPhase(PhaseList  $\cup$  { $p$ })
else
  PhaseElect' = SelectPhase(PhaseList - { $p$ })

```

-
- t_{event} *abort-phase*(p) O :
 - preconditions:
 $(RunningP', ase=nullphase) \wedge (Phase(PhaseElect)=p)$
 $\wedge (Mode(PhaseElect)=abort)$
 - postconditions:
 $ExecMode'(p) = abort$
 $ResumeTime'(p) = t_{event}$
 $ResourceRequested'(p)=0$;cancel attempt to acquire more resources
 $RunningPhase'=Phase(PhaseElect)$
 - t_{event} *preempt-phase*(p) S :
 - preconditions:
 $(RunningPhase=p) \wedge (RunningPhase \neq nullphase)$
 $\wedge (RunningPhase \neq Phase(PhaseElect))$
 - postconditions:
 if $(ExecMode(p) = normal)$ then
 $ExecClock'(p) = ExecClock(p) - (t_{event} - ResumeTime(p))$
 else
 $AbortClock'(p) = AbortClock(p) - (t_{event} - ResumeTime(p))$

 ;note p is not resource-waiting
 $RunningPhase'=nullphase$
 - t_{event} *resume-phase*(p) S :
 - preconditions:
 $(RunningPhase=nullphase) \wedge (Phase(PhaseElect)=p)$
 $\wedge (Phase(PhaseElect) \neq nullphase) \wedge (Mode(PhaseElect)=normal)$
 $\wedge \neg ResourceWaiting(Phase(PhaseElect))$
 - postconditions:
 $ResumeTime'(p) = t_{event}$
 $RunningPhase'=Phase(PhaseElect)$
-

Figure 3-4: Other Phase Operations Accepted by DASA Scheduling Automaton

activity begins with a 'request-phase' operation that declares its needs for its initial computational phase. Subsequent 'request-phase' events mark the end of one computational phase and the beginning of another. A final 'request-phase' operation denotes the completion of the activity's last computational phase. Of course, simple activities may consist of only one or possibly a few computational phases.

The precondition for accepting a 'request-phase' operation is simply *true*. That is, a 'request-phase' operation can be accepted at any time under any circumstances. This arrangement allows new phases to arrive at any instant, thus permitting activities to be submitted to the automaton asynchronously, just as they would be if they were initiated in response to interrupts.

The two arguments associated with each 'request-phase' event serve to specify the anticipated needs of the new computational phase: (1) v , the time-value function defining the value to the application of

-
- $t_{event} request(r) p$:
 - preconditions:
 $(RunningPhase=p) \wedge (RunningPhase \neq nullphase)$
 - postconditions:
 $ExecClock'(p) = ExecClock(p) - (t_{event} - ResumeTime(p))$
 $ResourceRequested'(p)=r$; indicate p is resource-waiting
 $PhaseElect' = SelectPhase(PhaseList)$
 $RunningPhase=nullphase$; give up processor until 'granted' resource
 - $t_{event} grant(p, r, undotime(r)) S$:
 - preconditions:
 $(RunningPhase=nullphase) \wedge (Phase(PhaseElect)=p) \wedge (r \neq nullresource)$
 $\wedge (ResourceRequested(Phase(PhaseElect))=r)$
 $\wedge (Mode(PhaseElect)=normal)$
 - postconditions:
 $ResumeTime'(p) = t_{event}$
 $AbortClock'(p) = AbortClock(p) + undotime(r)$
 $RunningPhase' = Phase(PhaseElect)$
 $Owner'(r)=p$; indicate ' p ' is owner of resource
 $ResourceRequested'(p)=\emptyset$
 $ResourcesHeld'(p)=ResourcesHeld-r$
-

Figure 3-5: Resource Operations Accepted by DASA Scheduling Automaton

completing the phase at any instant in time; and (2) $t_{expected}$, the amount of computation time that would be expected to execute the phase if there were no contention for shared resources — including the processor.

In addition, there is no indication about the shared resources that will be needed by the phase. This reflects the belief, explained earlier, that in order to allow a potentially high degree of concurrency, it may often be necessary to use techniques that preclude the exact knowledge of which resources will be needed by a computational phase.

The 'request-phase' operation has the longest set of postconditions of any of the operations accepted by the DASA Scheduling Automaton. This is due in large part to the fact that the postconditions handle the conclusion of one computational phase and the initiation of another. If the currently executing phase issues the 'request-phase' operation, then the operation marks a transition between phases. In that case, the value accrued by completing the phase is added to the running total for the application, and any shared resources held by the activity are released. (Note that if the activity had been aborting the computational phase, no value would be gained by completing the phase, since that simply represents the completion of the abort.)

If the activity that issued the 'request-phase' operation was not executing at that time, then it is a new activity. There is no previous phase to handle in that case.

Whether or not the computational phase is the first for the activity, the 'request-phase' postconditions

dictate that the time-value function and expected compute time parameter are associated with the new phase. The expected compute time parameter is used to initialize a virtual clock, called *ExecClock*. This clock indicates the amount of time required to complete the current phase for a given activity.

Other state components are altered as well. *AbortClock* is similar to *ExecClock* — it indicates the amount of time required to abort the current phase of an activity. Each time a new shared resource is acquired during a phase, *AbortClock* is increased by a resource-specific amount of time. Initially, it takes no time to abort a computational phase since nothing has been done yet and no shared resources have been acquired. Furthermore, *ExecMode* for the new phase is 'normal', not 'abort'.

It is possible that the 'request-phase' event may signal the completion of the final phase of an activity. In that case, the required computation time, $t_{expected}$, is declared to be zero — that is, no more computational cycles are needed for the activity.

If the 'request-phase' event does mark the completion of processing for an activity, then the phase is removed from the list of known phases, *PhaseList*. Otherwise, the phase is a member of *PhaseList*.

Finally, *SelectPhase()* is consulted to decide which phase should be executed now. Furthermore, if the currently executing activity (*RunningPhase*) issued the 'request-phase' event, it surrenders the processor — clearing the way to execute the *PhaseElect* specified by *SelectPhase()*. (Note that this really has no effect if *PhaseElect* is part of the currently executing activity. In that case, while the processor will nominally begin executing the *nullphase*, it will actually resume execution of the *PhaseElect* immediately. The transition to the *nullphase* is only a convenience in terms of modeling the automaton. After reviewing the other scheduling events accepted by the DASA Scheduling Automaton, the convention employed throughout to mark potential changes in execution due to a preemption, abortion, or unsatisfiable request should be clear.)

Abort-Phase. As modeled, phases are aborted only as a result of a decision by the scheduling function, *SelectPhase()*²¹.

By convention, each time the executing activity, *RunningPhase*, makes a new request to either begin a new phase or to acquire a new shared resource — necessitating a scheduling decision — the activity gives up the processor. That is, as a postcondition for accepting one of these requests, *RunningPhase* is set to be *nullphase*. This is done to meet the preconditions to accept either an 'abort-phase' or a 'resume-phase' event. Once the processor is idle, then if the execution mode of *PhaseElect* is 'abort', then an 'abort-phase' event can be accepted by the DASA Scheduling Automaton.

The postconditions for this event make sure that the phase is aborting, note the time at which execution

²¹This should not be viewed as precluding the possibility of an activity aborting a phase autonomously — perhaps due to a failure within a transaction. Rather, the model can easily be extended to accommodate that possibility: If the executing activity decides to abort the current phase, it issues an 'abort-self' event. This event changes the execution mode of the phase to 'abort', consults *SelectPhase()* to determine what to run next, and gives up the processor. When the scheduler selected that phase to begin its abort processing, it would issue an 'abort-phase' event, and processing would continue as described above.

resumed, cancel any outstanding requests for shared resources (since no new resources must be acquired to undo whatever was done to those previously acquired), and designate the new executing phase.

Preempt-Phase. As indicated by its precondition, the scheduler issues a 'preempt-phase' event if the processor is executing some phase other than the *PhaseElect* or the *nullphase*. In response, the current *RunningPhase* is suspended, its execution clock (either *ExecClock* or *AbortClock*, depending on the execution mode) is updated to reflect the true time left to free the shared resources held by the phase, and the processor is left idle.

Of course, the processor will probably not remain idle for long since either an 'abort-phase', a 'resume-phase', or a 'grant' event will be issued to execute another phase: (1) an 'abort-phase' event is issued for a phase that is being aborted, (2) a 'resume-phase' event is issued for a phase that is executing normally, but is not waiting for a resource (that is, it is a previously preempted phase); and (3) a 'grant' event is issued for a phase that is executing normally and is waiting for access to a shared resource. All three of these scheduling events require that the processor be idle before they dispatch the next phase. (Along with the 'preempt-phase' event, the 'request-phase' and 'request' events also leave the processor idle when appropriate to set the stage for these phase-dispatching events.)

Resume-Phase. The 'resume-phase' event resumes the execution of a previously preempted phase. The processor must be idle before a 'resume-phase' event can be accepted by the DASA Scheduling Automaton, and the phase resumed must be executing normally — as opposed to aborting — and must not be waiting on access to a shared resource.

The postconditions for the acceptance of a 'resume-phase' event note the time at which execution of the phase resumed and assign the processor to execute the phase.

Request. A 'request' event signals that the currently executing phase wishes to access a shared resource. As denoted by the event's preconditions, such a request can be made at any time while the phase is executing on the processor.

After accepting a 'request' event, the postconditions for the event update the requesting phase's execution clock to indicate the exact time left to complete the phase, record the resource that has been requested by the phase, select the next phase to be executed (possibly the requesting phase), and remove the requesting phase from the processor.

It should be understood that the decision to suspend the requesting phase's execution is only made to provide a simple, coherent formal model, not to suggest the actual design of an implementation of the DASA algorithm. Notice, for example, that in the formal model, it is quite possible that a phase could request a resource that is currently available, give up the processor, and immediately be reassigned the processor as the result of a 'grant' event. This is perfectly fine in the model, but an efficient implementation of the algorithm should decide whether the processor should actually be turned over to another phase before ever suspending execution of the current phase.

Grant. The 'grant' scheduling event assigns the processor, which must be idle, to execute a phase that has been blocked awaiting access to a shared resource. The phase assigned to execute has been previously selected and is designated *PhaseElect*.

Once the 'grant' event has been accepted by the DASA Scheduling Automaton, the postconditions associated with that event record the time at which the phase is granted the resource, adjusts the *AbortClock* to indicate the increment in work that is required to undo actions on the newly acquired shared resource, manipulates various relations to show that the resource now belongs to the designated phase, and starts the processor executing that phase.

Although there are 'request' and 'grant' events, there is no explicit 'release' event. This is due to the model of computation that has been adopted. Since all activities are composed of a sequence of computational phases and all shared resources that are acquired during a phase are released at the completion of the phase, there is no need for such an event. Rather, an implicit release of these resources is performed as part of the 'request-phase' event, which, among other things, denotes the completion of a phase (as described above).

3.2.1.3. 'SelectPhase' Function for DASA Automaton

The function 'SelectPhase()' embodies the DASA scheduling algorithm. As shown in Section 3.2.1.2, *SelectPhase()* is evaluated each time a 'request-phase' or a 'request' event is encountered. In Figure 3-6, *SelectPhase()* is formally defined as a mathematical function. Since this definition looks quite different than the brief procedural definition offered in Section 3.1.3, a few comments are in order to explain the utility of this format and its organization.

The algorithm is described as a mathematical function for a few reasons. First and foremost, it is a concise and precise notation. But it also is more expressive in some ways than procedural definitions. Specifically, this mathematical format is capable of expressing the sequential nature of a set of operations — by using functional composition, for example, where each function corresponds to one of the sequential operations. At the same time, this mathematical format can also express the *nondeterminism* that is present in the algorithm definition. For instance, the order in which the elements in a list are examined may or may not be important. When the order is important, there is a specific method to describe the order. This is said to be *deterministic*, in that there is only one correct order. When the order is unimportant, any order will do; and so this case is said to be *nondeterministic*. A typical procedural definition cannot readily capture this nondeterminism. Such a definition would usually have to specify some ordering, even if the ordering was not critical.

The function 'SelectPhase()', when given a list of phases, selects the next phase to run and specifies its execution mode (either 'normal' or 'abort'). Informally, the definition shown for 'SelectPhase()' in Figure 3-6 determines a set of phases that can feasibly meet their time constraints given all of the information that is currently known about them. It then selects one of the phases from this set that must be done by the earliest deadline and designates it as the next phase that the processor should execute.

All of the phases in the phase list P that was passed to *SelectPhase()* are considered when constructing the list of phases that can feasibly execute. Also, as each phase is examined in turn, any dependency that prevents it from executing immediately are noted and resolved by indicating those other activities that must precede it in any schedule — either completing or aborting their current phases.

To see how the definition actually specifies the desired behavior, a closer look is necessary. Towards that end, consider constructing the definition from the bottom up. While a few of the functions appearing in the definition have already been discussed briefly, others are totally new.

The following descriptions constitute an informal definition of the functions comprising *SelectPhase()*. Often only the "main" or "normal" case value will be discussed for a function, even though its definition includes a number of other cases as well²². This is because the other cases usually handle degenerate situations that arise as a result of the recursive nature of some of the function definitions.

To start, remember that a few basic functions were described in Section 2.3.1. They include *Deadline()* and *Val()* and are used in the definitions that follow as basic building blocks.

Also remember that *SelectPhase()* is a function that is evaluated within the context of the DASA scheduling automaton. As such, it has access to all of the state components of the automaton, which in turn provides access to all of the status information for each phase in the system. Furthermore, since *SelectPhase()* is always evaluated as a result of accepting a scheduling operation, the t_{event} that appears in the formal definition refers to the timestamp for the accepted event.

With that background in mind, we can begin to examine the formal definition of *SelectPhase()* in earnest. Consider first the set of functions that form the dependency lists and evaluate the potential value densities of all of the phases in the system.

The function 'Dep()', evaluated for a specified phase, returns as its value the phase that is currently preventing the specified phase from executing (due to a dependency). If the phase is ready to execute immediately, then the 'nullphase' is the value of 'Dep()'. Otherwise, the phase has requested a shared resource and is dependent on the owner of that resource — that is the phase that currently holds the shared resource — if there is one. The phase holding the resource must relinquish it before the dependent phase can continue execution.

The resource can be relinquished in one of two ways: either the phase can complete its normal course of execution or it can be aborted. Both of these alternatives take time²³, and the DASA algorithm attempts to minimize the amount of time waiting for the resource. So DASA completes the phase unless it is faster to abort it.

²²Which case is to be used to evaluate the function typically depends on the value of one or more arguments to the function.

²³As was pointed out earlier, a phase that has been aborted does not instantaneously return the shared resources allocated to it to the system. Rather, the shared resources must be placed into a meaningful, acceptable, safe, and (possibly) consistent state prior to their release. It is the processing that puts the shared resources into these acceptable states that consumes time after an abort has been issued for the phase.

$SelectPhase(P) =$
 $\quad pickone(mustfinishby(DL_{first}(mpplist), P_{scheduled}(P))),$
 $\quad \text{where}$
 $\quad mpplist = tobescheduled(P_{scheduled}(P))$

$pickone(MPP) =$
 $\quad \langle normal, p \rangle,$ if $\langle normal, p \rangle \in MPP \wedge Dep(p) = nullphase$
 $\quad \langle abort, p \rangle,$ if $\langle abort, p \rangle \in MPP$
 $\wedge \neg (\exists q) (\langle normal, q \rangle \in MPP$
 $\wedge Dep(q) = nullphase)$
 $\quad \langle normal, nullphase \rangle,$ otherwise

$DL_{first}(MPP) =$
 $\quad \infty,$ if $MPP = \emptyset$
 $\quad Deadline(p) \mid (\langle normal, p \rangle \in MPP)$
 $\quad \wedge (\forall q) (\langle normal, q \rangle \in MPP \rightarrow Deadline(q) \geq Deadline(p)),$
otherwise

$P_{scheduled}(P) =$
 $\quad \emptyset,$ if $P = \emptyset$
 $\quad P_{feasible}(P_{scheduled}(P - \{p\}) \cup \{p\}),$ if $p \in P_{leastPV}(P)$

$P_{feasible}(P) =$
 $\quad \emptyset,$ if $P = \emptyset$
 $\quad P,$ if $feasible(P)$
 $\quad P_{feasible}(P - \{p\}),$ where $p \in P_{leastPV}(P),$ otherwise

$P_{leastPV}(P) =$
 $\quad \emptyset,$ if $P = \emptyset$
 $\quad \{p \mid (p \in P)$
 $\quad \wedge (\forall q) (q \in P \rightarrow ((PVD(p) \leq PVD(q))$
 $\quad \wedge (PVD(p) = PVD(q) \rightarrow ExecClock(p) \leq ExecClock(q))))\},$
otherwise

$tobescheduled(P) =$
 $\quad \emptyset,$ if $P = \emptyset$
 $\quad \{\langle normal, p \rangle\} \cup dependencylist(p) \cup tobescheduled(P - \{p\}),$
if $p \in P$

$dependencylist(p) =$
 $\quad \emptyset,$ if $Dep(p) = nullphase$
 $\quad dependencylist(Dep(p)) \cup \{\langle normal, Dep(p) \rangle\},$
if $AbortClock(Dep(p)) \geq ExecClock(Dep(p))$
 $\quad \{\langle abort, Dep(p) \rangle\},$ otherwise

Figure 3-6: Functional Form of DASA Algorithm

$mustcompleteby(t,P) =$
 0, if $t < t_{event}$
 $\{p \mid \langle normal, p \rangle \in tobescheduled(P) \wedge Deadline(p) \leq t\}$, otherwise

$mustfinishby(t,P) =$
 0, if $P = \emptyset \vee t < t_{event} \vee mustcompleteby(t,P) = \emptyset$
 $reduce(t, P, \{\langle normal, p \rangle\} \cup dependencylist(p) \cup mustfinishby(t, P - \{p\}))$, if $p \in mustcompleteby(t,P)$
 if $p \in mustcompleteby(t,P)$

$reduce(t,P,MPP) =$
 $reduce(t, P, MPP - \{\langle abort, p \rangle\})$, if $\langle abort, p \rangle, \langle normal, p \rangle \in MPP$
 if $\langle abort, p \rangle \in mustfinishby(t, P)$
 MPP, otherwise

$feasible(P) = true$, iff $(\forall t)[(t \geq t_{event}) \rightarrow timerequiredby(mustfinishby(t,P)) \leq (t - t_{event})]$

$timerequiredby(MPP) =$
 0, if $MPP = \emptyset$
 $ExecClock(p) + timerequiredby(MPP - \{\langle normal, p \rangle\})$, if $\langle normal, p \rangle \in MPP$
 $AbortClock(p) + timerequiredby(MPP - \{\langle abort, p \rangle\})$, if $\langle abort, p \rangle \in MPP$

$PVD(p) =$ 0, if $ExecMode(p) = abort$
 $\frac{Val(p) + PV(Dep(p))}{ExecClock(p) + PT(Dep(p))}$, otherwise

$PV(p) =$ 0, if $p = nullphase$
 0, if $AbortClock(p) < ExecClock(p)$
 $Val(p) + PV(Dep(p))$, otherwise

$PT(p) =$ 0, if $p = nullphase$
 $AbortClock(p)$, if $AbortClock(p) < ExecClock(p)$
 $ExecClock(p) + PT(Dep(p))$, otherwise

$Dep(p) =$ nullphase, if $ResourceRequested(p) = nullresource$
 Owner(ResourceRequested(p)), otherwise

Figure 3-6: Functional Form of DASA Algorithm, *continued*

The function 'dependencylist()' uses the information supplied by 'Dep()' about the dependencies of individual phases to construct a list that includes all of the phases that must execute before a specified phase. 'Dependencylist()' also specifies the execution mode for each of the phases that must be executed prior to the specified phase. Therefore, the dependency list is actually a set of mode-phase pairs of the form *<mode,phase>*. It is in this function that the decision to minimize the length of time to remove dependencies is implemented.

The definition of the function is recursive. It initially examines the phase, *p*, that was given as its argument. If *p* is not dependent on any other phase, then its dependency list is empty. Otherwise, it will be non-empty. Specifically, if it is faster to abort the phase on which *p* depends, then the dependency list will have only one member: *<abort,Dep(p)>*. Alternatively, if it is at least as fast to complete the normal execution of the phase on which *p* depends, then *p*'s dependency list will be constructed by adding *<normal,Dep(p)>* to the dependency list of *Dep(p)*.

Once a dependency list has been determined for a phase, it is possible to evaluate the potential value density for that phase. This is done by the function *PVD()*, which employs two auxiliary functions, *PV()* and *PT()*. These functions are similar to those discussed earlier in this chapter, in Section 3.1.3. They total the value that may be accrued and the execution time that is required jointly by the given phase and all of the phases in its dependency list. (Note that aborting a phase requires time but yields no value directly.) These totals are then used to determine the potential value density for the specified phase.

The function *P_{leastPV}()* examines a set of phases and returns the subset of phases that have the lowest potential value. In case more than one phase has the same (lowest) potential value density, the phase or phases that will consume the least execution time is returned. This choice is made because, when considering two phases with the same PVD, the phase that executes longer will obtain a higher value than the one that runs shorter since value is the product of PVD and execution time.

Another group of functions determine the amount of time required to carry out a specified set of executions and aborts over all of the critical time intervals, thereby allowing the feasibility of the specified computations to be ascertained. So, for instance:

- *timerequiredby()* — given a set of mode-phase pairs, this function determines the total execution time required to carry out all of the specified computations;
- *mustcompleteby()* — given a time and a set of phases, this function identifies those phases that must complete execution by the specified time;
- *mustfinishby()* — given a time and a set of phases, this function identifies all of the normal executions and abortions that must finish by the specified time; whereas, *mustcompleteby()* identified those phases that had to complete their normal executions by the specified time, *mustfinishby()* adds to that group all of the other work that must be done in order to remove any existing dependencies that might prevent those phases from executing immediately; also notice that this function uses another function, *reduce()*, to eliminate unnecessary aborts from the resultant list;
- *reduce()* — this function eliminates unnecessary aborts by noticing cases where the same

phase is being both completed and aborted²⁴, but the completion must be done prior to the abort due to the dependencies currently in effect; of course, there is no need to abort a phase once it has completed;

- *feasible()* — given a set of phases, this function determines whether all of the phases in the set, along with all of the other computations on which they depend, can meet their deadlines; for a schedule to be feasible, at every point in time the total amount of time required to complete the computations that must be done by that time must never exceed the actual time remaining until that time.

With this set of functions to use as building blocks, it is possible to describe at a fairly high level how to select the phase that should execute next.

A set of phases that can be feasibly run (given current knowledge of requirements and resources) is constructed by examining each existing phase ordered by PVD, starting with the phase with the highest PVD. The functions $P_{scheduled}()$ and $P_{feasible}()$ construct this set²⁵. Given a set of N phases, $P_{scheduled}()$ will first (recursively) determine which of the $N-1$ phases with the greatest potential value may feasibly be executed. $P_{scheduled}()$, using $P_{feasible}()$ and ultimately *feasible()*, then determines if the phase with the least potential value can feasibly be added to the set. If so, it is.

Once $P_{scheduled}()$ has identified which phases can be completed successfully, it is fairly straightforward to determine which phase should be executed first. The auxiliary function $DL_{first}()$ specifies the earliest deadline that must be met by those phases that can complete execution. That information, along with the set of phases to be completed, is once again passed to the function *mustfinishby()* to determine all of the work that must be done by the earliest deadline. And finally, *pickone()* selects a mode-phase pair from that set to execute first. *pickone()* always prefers to complete a phase normally if possible, but if that cannot be done, it will initiate (or continue) the abortion of a phase.

3.2.2. Observations on the Definition

Several observations can be made now that the formal definition of the DASA Scheduling Automaton has been presented in full. Each of the following sections focus on an interesting observation.

3.2.2.1. Manifestation of Desirable Properties

Section 3.1.1 listed five desirable properties that the DASA algorithm should possess. Now that the algorithm has been presented in some depth, those properties should be reviewed again:

1. explicitly account for dependencies — this has been accomplished. The definition of *SelectPhase()* was described from the bottom up, and the first thing that was done in considering any phase was to determine those phases that it depends on (its dependency list) and the aggregate value of this group of phases to the application.

²⁴It is not unexpected that both the completion and the abortion of a single phase will sometimes be executed. In the expected case, the phase is aborted in order to allow some other phase, with a tighter deadline, to execute. Later, the aborted phase can be restarted and completed normally, still meeting its time constraint.

²⁵Note that the functions that are named $P_x()$ all represent sets of phases.

2. **minimize effort** — this property refers to the amount of effort required to enable a phase to be ready to execute. The DASA algorithm has minimized this effort by minimizing the time needed to eliminate each of the dependencies for that phase: if it is quicker to abort a phase than it is to execute it to completion, then it is aborted. This minimizes a latency, of sorts, at the possible cost of later reexecuting phases that have been aborted.
3. **maximize return/benefit** — the use of the potential value density addresses this concern directly. As outlined in Section 3.1.1, by adding those phase groups (a phase along with the phases that comprise its dependency list) with the highest PVD to the schedule first, the algorithm guarantees that no other phase group can attain a higher aggregate value consuming the same number of cycles, based on current knowledge.
4. **maximize the chance of meeting deadlines** — this property has been met through the placement of phases in the tentative schedule that is recursively constructed by *SelectPhase()*. The key observation is that, although phases are considered for addition to the tentative schedule in order of decreasing PVD, they are actually added to the schedule in an order that is determined only by the deadlines of the phases being placed and their dependencies: stated informally, a phase that is to be executed to completion is inserted in the schedule according to its deadline, unless that time is too late to allow a scheduled phase that depends on it to complete in time. In the latter case, it inherits the latest deadline that will allow the dependent phase to meet its deadline.
5. **globally optimize schedule** — the function *reduce()* applies some global reductions to the tentative schedule that is recursively constructed by *SelectPhase()*. This is necessary since each phase is added to the schedule, along with its dependencies, independently of any other phases that may already be part of the schedule. As a result, it is possible that the abortion of a phase may be scheduled after the same phase's completion. Although this would have no real effect on the sequence of phases executed — after the phase had completed, it would release all of the shared resources it was holding so that the next evaluation of *SelectPhase()* would have no dependency requiring its abortion — it is important to eliminate it from the tentative schedule so that the most realistic estimate of processor cycle demands can be maintained.

3.2.2.2. Nondeterminism in Definition

As was mentioned in Section 3.2.1.3, a mathematical form was chosen for the function definitions in part to allow orderings to be specified when they are important, and to be unspecified otherwise. The definitions of *SelectPhase()* and its subsidiary functions provide examples of each:

- Order matters when determining which phases to add to the tentative schedule. The function *P_{scheduled}()* selects the phase to be removed from the set *P* it was given according to the PVDs and execution clocks of the individual phases in *P*. (Even here there is some nondeterminism, since it is possible — though probably unlikely — for more than one phase to belong to the set *P_{leastPV}()*, with each of these phases having the same PVD and execution clock value.)
- Order does not matter when the set of mode-phase pairs that must be in a schedule in order to successfully complete a given set of phases is constructed. This construction is carried out by the function *toBescheduled()*, and in this case, the phase to be removed from the set *P* for the next recursive call to *toBescheduled()* is totally unspecified — any element of *P* will do.

There are other examples for each of these cases in the DASA definition, but these serve to illustrate the ability of the notation to capture the essential aspects of ordering without imposing unnecessary constraints. This clarity may be of considerable benefit when weighing the correctness of alternative implementations of the algorithm that use different orderings for various evaluations.

3.2.2.3. Explicit Appearance of Time

Time doesn't explicitly appear in many of the individual function definitions. This may be unexpected for an environment where time — and meeting time constraints — is a central concern. Of necessity, time explicitly plays a role in testing the feasibility of executing groups of phases. And while this testing occurs throughout the evaluation of *SelectPhase()*, references to time seem infrequent since phases are added to the tentative schedule according to their potential value density, not according to the urgency of their time constraints.

3.3. Scheduling Example Revisited

Now that the scheduling algorithm has been presented, it is possible to reconsider the scheduling example discussed in Section 1.3. Once again, the problem is to schedule phases p_a , p_b , and p_c so as to meet their time constraints, if possible. In fact, it is possible, and this is shown by the bottom execution profile in Figure 3-7. Notice that phase p_a is aborted during the course of execution, thus allowing phase p_b to meet its deadline. This necessitates the reexecution of the start of phase p_a at a later time.

The top of Figure 3-7 shows the execution profile for a scheduler that is identical to DASA, except that it cannot abort phases. It, too, meets all of the deadlines, while consuming fewer cycles than DASA in the process. However, it must tolerate a longer delay between the time that it determines that a given phase should be executed and the time at which that phase may actually begin execution due to existing dependencies. This variant of the DASA algorithm is shown only as a reference point. At this point, it is not anticipated that it will be studied in significant depth as part of the proposed thesis research.

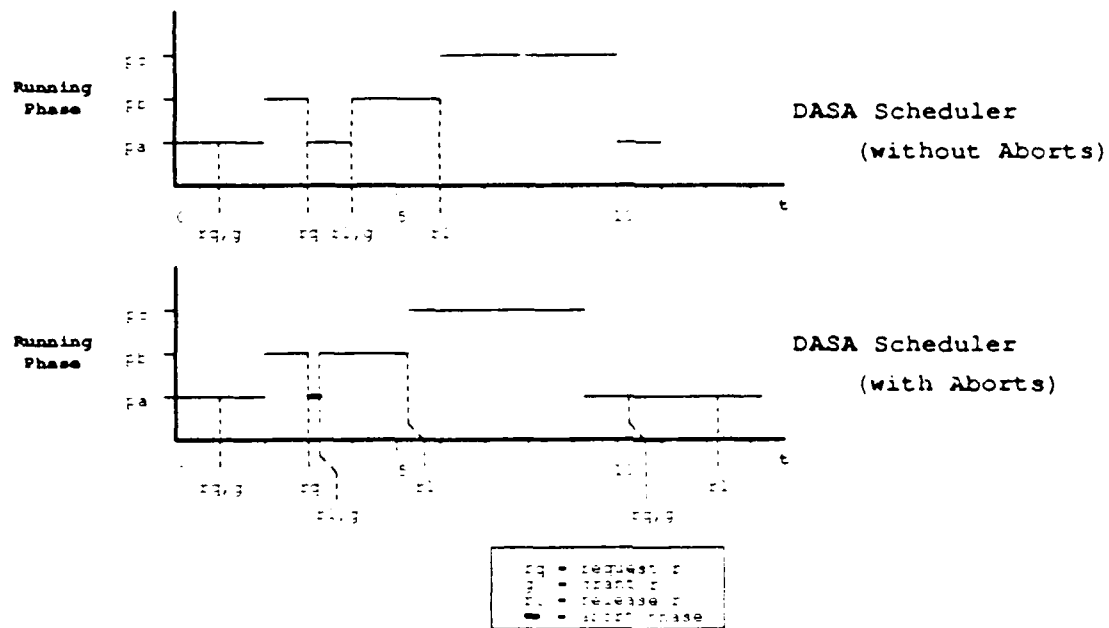


Figure 3-7: Execution Profiles for DASA Scheduler with and without Aborts

Chapter 4

Analytic Results

This chapter presents a set of analytic results that argue for the benefits of the DASA algorithm. First, a number of high-level requirements that real-time scheduling algorithms must possess is discussed. Then a strategy for demonstrating that the DASA scheduling algorithm possess those properties is outlined, followed by a set of proofs conforming to that strategy. The final section of the chapter discusses various interesting behaviors that the DASA algorithm may demonstrate, which are revealed by its formal description.

4.1. Requirements for Scheduling Algorithms

Any practical solution to the problem of scheduling while taking dependencies into account must be correct, valuable, and tractable.

The solution must be *correct*. Specifically, any scheduling decisions that are made must observe all of the known dependencies. Therefore, for instance, any activity that is selected to execute must be able to execute at that point in time. The solution must also obey the concurrency control rules of the model; in particular, for the model presented here, mutually exclusive access to the shared resources must be guaranteed.

The solution must be *valuable*. When cast in the computational model described above, this requirement simply means that the schedules dictated by the scheduler must yield good values relative to other scheduling algorithms. Notice that this is partially a comment on the scheduler's behavior in normal situations and partially a comment on its behavior in overload situations. In normal (non-overload) situations, the ordering of activities is critical and many schedulers will not order them appropriately, even when there are sufficient processor cycles present to satisfy all demands; in overload situations, the system/application should display a graceful degradation of function²⁶. Both of these types of situation are accurately gauged by the value metric previously introduced.

Finally, the solution must be computationally *tractable/efficient*. That is, the solution must consume, at worst, an amount of time and space that is polynomial in the problem size — in this case, the problem's size is the number of phases under consideration by the scheduler.

²⁶Even schedulers that take dependencies into account may handle overload situations differently, resulting in different scheduling decisions, and hence different values, for executing the application.

4.2. Strategy for Demonstrating Requirement Satisfaction

Analytic proofs have been constructed to demonstrate the correctness, value, and tractability of the DASA algorithm. These proofs are contained in Section 4.3.

To demonstrate correctness, it is shown that the DASA algorithm respects any existing dependencies among phases and makes legal selections. This is accomplished by demonstrating that any phase that DASA selects for execution is capable of executing immediately. That is, it is shown that DASA will either (1) select a phase that is ready to run (i.e., is not blocked), or (2) designate that a phase is to be aborted, which can be executed immediately for any phase, blocked or ready to run. This proof is presented in Section 4.3.1.

To demonstrate value, proofs serve to illustrate that DASA performs well when compared to other scheduling algorithms in appropriate situations. In particular, when there are no dependency considerations, DASA can be compared to a number of well-known algorithms. In fact, it is shown that, if there are no overload conditions, the DASA automaton will accept the same histories as an automaton that accepts histories conforming to Locke's Best Effort Scheduling Algorithm (LBESA). Not coincidentally, this is simply a deadline-ordered history. In overload situations, it is demonstrated that the DASA automaton will accept histories that the LBESA automaton will not accept, and that these histories may have a higher value than any history that the LBESA automaton may accept involving the same phases with the same scheduling parameters. These proofs are presented in Section 4.3.2.

To demonstrate tractability, a procedural version of the DASA algorithm has been developed, and its complexity has been analyzed to prove that the time and space requirements of the algorithm are indeed polynomial in problem size — that is, that the time and space required to execute the algorithm are each proportional to the number of active phases raised to some polynomial power. Both the procedural version of the DASA algorithm and the derivation of its space and time properties are presented in Section 4.3.3.

4.3. Proofs of Properties

The proofs in the sections that follow demonstrate properties of the DASA scheduling algorithm according to the strategy outlined in the preceding section. Each section contains all of the proofs corresponding to a single property of concern. In addition to the proofs themselves, other material that must be developed to complete the proofs is also presented. For example, in Section 4.3.2.1, a derivation of another scheduling automaton is presented. This automaton is subsequently used in proofs to assess the utility of the DASA algorithm.

4.3.1. Algorithm Correctness

There is only one proof in this section. It demonstrates that DASA respects all existing dependencies among phases by showing that the phase selected for execution can execute immediately. Therefore, no phase is ever selected for normal execution if it is dependent on some other execution. Of course, a phase that is blocked due to a dependency could be selected to abort, since it can abort at any time regardless of dependency considerations.

4.3.1.1. Proof: Selected Phases May Execute Immediately

Theorem 1: Given *PhaseList*, the set of phases known to the DASA automaton, prove that the phase selected for execution, *PhaseElect*, is eligible to run at that point.

Proof. In every case in the DASA automaton, *PhaseElect*, the phase selected for execution, is determined by evaluating *SelectPhase(PhaseList)*. The function *SelectPhase()* is defined as:

$$\begin{aligned} \text{SelectPhase}(P) = & \text{pickone}(\text{mustfinishby}(\text{DL}_{\text{first}}(\text{pmplist}), P_{\text{scheduled}}(P))), \\ & \text{where} \\ & \text{pmplist} = \text{tobescheduled}(P_{\text{scheduled}}(P)) \end{aligned}$$

and *pickone()* is defined as.

$$\begin{aligned} \text{pickone}(PMP) = & \begin{aligned} & \langle \text{normal}, p \rangle, & \text{if } \langle \text{normal}, p \rangle \in PMP \\ & \quad \wedge \text{Dep}(p) = \text{nullphase} \\ & \langle \text{abort}, p \rangle, & \text{if } \langle \text{abort}, p \rangle \in PMP \\ & \quad \wedge \neg (\exists q) (\langle \text{normal}, q \rangle \in PMP \\ & \quad \quad \wedge \text{Dep}(q) = \text{nullphase}) \\ & \langle \text{normal}, \text{nullphase} \rangle, & \text{otherwise} \end{aligned} \end{aligned}$$

Notice that *pickone()* will return one of three values:

- $\langle \text{normal}, p \rangle$, for some phase p — this occurs only when $\text{Dep}(p) = \text{nullphase}$; in that case, p is ready to run by definition;
- $\langle \text{abort}, p \rangle$, for some phase p — any phase may be aborted at any time, even if it had previously been waiting to access a shared resource; so once again, by definition, p is ready to run;
- $\langle \text{normal}, \text{nullphase} \rangle$ — this designates an idling condition, which is always possible, so nullphase is trivially ready to run²⁷.

In each case, *PhaseElect* is assigned a phase/mode pair in which the phase is ready to run.

EndOfProof

²⁷Notice that $\langle \text{normal}, \text{nullphase} \rangle$ is returned only in the case that there are no phases ready to run in either their normal mode or their abort mode.

4.3.2. Algorithm Value

Since most scheduling algorithms do not utilize dependency information, it is difficult to make fair comparisons between their performance and that of DASA when dependencies are involved. Therefore, this section will compare DASA to another algorithm (LBESA) in the absence of any dependencies.

Since LBESA was shown to outperform a number of standard algorithms in a range of situations, a favorable comparison with LBESA will demonstrate that DASA behaves well.

To that end, the two proofs presented in this section demonstrate that the DASA algorithm performs well when compared to the LBESA algorithm. They consider a set of activities that are independent of one another, each of which is described by a time-value function that is a step function. They show:

1. If there is no overload, then both DASA and LBESA yield identical expected value to the application.
2. Under overload, DASA may schedule more activities than LBESA, yielding a greater expected value than LBESA.

Before presenting the two proofs, the next two sections develop the formal scheduling automata that they will use. First, Section 4.3.2.1 presents the LBESA Scheduling Automaton. Then Section 4.3.2.2 presents a scheduling automaton corresponding to the DASA algorithm when there are no dependencies to consider.

4.3.2.1. LBESA Scheduling Automaton

The LBESA Scheduling Automaton is cast using the General Scheduling Automaton Framework described in Section 2.3.2. Once again, each scheduling decision is made based on the set of phases currently known to the automaton: $\{p_0, p_1, p_2, \dots\}$.

LBESA Automaton State Components. The state components associated with the LBESA Scheduling Automaton are presented in Figure 4-1. They are simply the General State Components that every scheduling automaton contains, and they were described in detail in Section 2.3.2.9.

Operations Accepted by LBESA Automaton. The operations accepted by the LBESA automaton and their preconditions and postconditions are shown in Figure 4-2.

These are a somewhat simpler version of those presented in Section 3.2.1.2 for the DASA Scheduling Automaton. Most notably, there are no operations for dealing with resources — in particular, there are no 'request' and 'grant' operations. (Of course, in keeping with the General Scheduling Automaton Framework, these operations actually exist for the LBESA Scheduling Automaton. However, their preconditions are defined to be *false*, indicating that events with these operations can never be accepted by the LBESA Scheduling Automaton.) In addition, there are no postconditions for 'request-phase' to release previously acquired resources, and the precondition for 'resume-phase' is one term shorter.

The LBESA Scheduling Automaton does not accept 'abort-phase' operations either. This is because the LBESA scheduling algorithm does not abort activities or phases. Such aborts are not required because the activities are all assumed to be independent.

General State Components:

- ExecMode: PHASE $\rightarrow$ MODE (MODE is either 'normal' or 'abort')
- ExecClock: PHASE $\rightarrow$ VIRTUAL-TIME
- AbortClock: PHASE $\rightarrow$ VIRTUAL-TIME
- ResumeTime: PHASE $\rightarrow$ TIMESTAMP
- Value: PHASE $\rightarrow$ (TIMESTAMP $\rightarrow$ VALUE) (initially Value(t) = 0)
- Total: VALUE (initially '0')
- RunningPhase: PHASE (initially 'nullphase')
- PhaseElect: MODE $\times$ PHASE (initially '<normal, nullphase>')
- PhaseList: list of PHASE (initially 'o')

Algorithm-Specific State Components:

- None

Figure 4-1: State Components of LBESA Scheduling Automaton

When activities are not independent, then aborts must be introduced into the model. Notice that this does not mean that the scheduler must generate abort signals, but rather, that there must be a way to return shared resources to acceptable states before allowing other activities to acquire them and to return the aborted activity to a known state (presumably to handle an abort exception) if it is to have any chance at continuing normal execution.

References to the 'AbortClock' state component have been left in the postconditions for the 'preempt-phase' operation merely for convenience when comparing it to another automaton. Since aborts are never used, the clause that deals with the 'AbortClock' state component will never actually have an effect.

'SelectPhase' Function for LBESA Automaton. The function 'SelectPhase()' embodies the LBESA scheduling algorithm in this scheduling automaton, just as the identically-named function had done in the DASA Scheduling Automaton. Figure 4-3 shows the definition of this function.

Since Locke never employed such formalisms in his work, he never provided as rigorous a definition for his scheduling algorithm as the one shown here. And he certainly never provided a mathematical function corresponding to his definition. As a result, the definition shown here captures Locke's algorithm in this framework.

There are a number of ways of defining *SelectPhase()*, and the one chosen parallels the structure of the *SelectPhase()* function for the DASA Scheduling Automaton in order to facilitate comparisons between them.

• t_{event} request-phase($v, t_{expected}$) P :

preconditions:
 $true$ <No preconditions here so that interrupts and other new phases can occur at any time>

postconditions:
 if (RunningPhase = p) then
 if (ExecMode(p) = normal) then
 Total' = Total + Value(p)(t_{event})
 else
 ;no value for aborted phase
 ;release the resources acquired during the phase
 ;involves no action for this automaton
 Value'(p) = v
 ExecClock'(p) = $t_{expected}$
 AbortClock'(p) = 0
 ExecMode'(p) = normal
 ;note that p is not resource-waiting

 ;make sure p is part of the list of phases, if necessary
 if ($t_{expected} > 0$) then
 PhaseList' = PhaseList $\cup$ { p }
 else
 PhaseList' = PhaseList - { p }
 PhaseElect' = SelectPhase(PhaseList')
 if (p = RunningPhase) then
 ;give up processor until next 'resume-phase'
 RunningPhase = nullphase
 else
 ;happened under interrupt—leave 'RunningPhase' alone

• t_{event} preempt-phase(p) S :

preconditions:
 (RunningPhase = p) $\wedge$ (RunningPhase $\neq$ nullphase)
 $\wedge$ (RunningPhase $\neq$ Phase(PhaseElect))

postconditions:
 if (ExecMode(p) = normal) then
 ExecClock'(p) = ExecClock(p) - (t_{event} - ResumeTime(p))
 else
 AbortClock'(p) = AbortClock(p) - (t_{event} - ResumeTime(p))

 ;note p is not resource-waiting
 RunningPhase' = nullphase

• t_{event} resume-phase(p) S :

preconditions:
 (RunningPhase = nullphase) $\wedge$ (Phase(PhaseElect) = p)
 $\wedge$ (Phase(PhaseElect) $\neq$ nullphase) $\wedge$ (Mode(PhaseElect) = normal)

postconditions:
 ResumeTime'(p) = t_{event}
 RunningPhase' = Phase(PhaseElect)

Figure 4-2: Operations Accepted by LBESA Scheduling Automaton

$$\text{SelectPhase}(P) =$$

$$\text{pickone}(\text{mustfinishby}(DL_{\text{first}}(\text{pmplist}), P_{\text{scheduled}}(P))),$$

$$\text{where}$$

$$\text{pmplist} = \text{tobescheduled}(P_{\text{scheduled}}(P))$$

$$\text{pickone}(PMP) =$$

$$\langle \text{normal}, \text{nullphase} \rangle, \quad \text{if } PMP = \emptyset$$

$$\langle \text{normal}, p \rangle \mid \langle \text{normal}, p \rangle \in PMP, \quad \text{otherwise}$$

$$DL_{\text{first}}(PMP) =$$

$$\infty, \quad \text{if } PMP = \emptyset$$

$$\text{Deadline}(p) \mid (\langle \text{normal}, p \rangle \in PMP)$$

$$\wedge (\forall q)(\langle \text{normal}, q \rangle \in PMP \rightarrow \text{Deadline}(q) \geq \text{Deadline}(p)),$$

$$\text{otherwise}$$

$$P_{\text{scheduled}}(P) =$$

$$\emptyset, \quad \text{if } P = \emptyset$$

$$P_{\text{feasible}}(P_{\text{scheduled}}(P - \{p\}) \cup \{p\}), \quad \text{if } p \in P_{\text{lastDL}}(P)$$

$$P_{\text{feasible}}(P) = \emptyset, \quad \text{if } P = \emptyset$$

$$P, \quad \text{if } \text{feasible}(P)$$

$$P_{\text{feasible}}(P - \{p\}), \text{ where } p \in P_{\text{leastPV}}(P), \quad \text{otherwise}$$

$$P_{\text{lastDL}}(P) = \emptyset, \quad \text{if } P = \emptyset$$

$$\{p \mid (p \in P) \wedge (\forall q)[q \in P \rightarrow ((\text{Deadline}(p) \geq \text{Deadline}(q))$$

$$\wedge (\text{Deadline}(p) = \text{Deadline}(q) \rightarrow \text{PVD}(p) \leq \text{PVD}(q)))]\},$$

$$\text{otherwise}$$

$$P_{\text{leastPV}}(P) = \emptyset, \quad \text{if } P = \emptyset$$

$$\{p \mid (p \in P) \wedge (\forall q)[q \in P \rightarrow ((\text{PVD}(p) \leq \text{PVD}(q))$$

$$\wedge (\text{PVD}(p) = \text{PVD}(q) \rightarrow \text{ExecClock}(p) \leq \text{ExecClock}(q)))]\},$$

$$\text{otherwise}$$

$$\text{tobescheduled}(P) = \{\langle \text{normal}, p \rangle \mid p \in P\}$$

$$\text{mustcompleteby}(t, P) =$$

$$\emptyset, \quad \text{if } t < t_{\text{event}}$$

$$\{p \mid [p \in P \wedge \text{Deadline}(p) \leq t]\}, \quad \text{otherwise}$$

$$\text{mustfinishby}(t, P) =$$

$$\emptyset, \quad \text{if } P = \emptyset \vee t < t_{\text{event}}$$

$$\vee \text{mustcompleteby}(t, P) = \emptyset$$

$$\{\langle \text{normal}, p \rangle \mid p \in \text{mustcompleteby}(t, P)\}, \quad \text{otherwise}$$

$$\text{feasible}(P) = \text{true}, \text{ iff } (\forall t)[(t \geq t_{\text{event}}) \rightarrow \text{timerequiredby}(\text{mustfinishby}(t, P)) \leq (t - t_{\text{event}})]$$

$$\text{timerequiredby}(PMP) =$$

$$\emptyset, \quad \text{if } PMP = \emptyset$$

$$\text{ExecClock}(p) + \text{timerequiredby}(PMP - \{\langle \text{normal}, p \rangle\}), \text{ if } \langle \text{normal}, p \rangle \in PMP$$

$$\text{PVD}(p) = \text{VD}(p) = \frac{\text{Val}(p)}{\text{ExecClock}(p)}$$

Figure 4-3: Functional Form of LBESA Algorithm

Despite the degree to which the effort to cast these functions in the same form succeeded, there are still substantial differences between the two functions. The most important of these is the order in which phases are added to the tentative schedule by the two algorithms. This difference is seen in the $P_{\text{scheduled}}$ subsidiary function of each definition. LBESA adds phases to the tentative schedule in deadline order, nearest deadline first. DASA, on the other hand, adds phases to the tentative schedule in order of decreasing potential value density.

In the event that a tentative schedule is not feasible, both algorithms (effectively) remove phases from the tentative schedule in order of increasing value density or potential value density, respectively. The fact that LBESA adds phases to the schedule based on one attribute and sheds phases based on another, while DASA uses a single attribute for both purposes, causes the algorithms to make different scheduling decisions under certain circumstances. This leads directly to the fact that, under overload, DASA can attain greater value for an application than LBESA can, as is shown in Section 4.3.2.4.

Locke was silent on some details concerning his algorithm, such as which phase should be selected if two or more phases shared the nearest deadline in a schedule or which phase to shed if two or more phases had a common value density that was lower than that of all of the others phases in the tentative schedule. Whenever possible, these details have been resolved in the manner that seemed to make the most sense. For example, when two or more phases are characterized by the same value density, the phase requiring the least computation time is deemed to be less valuable than the others since its contribution to the overall value of the application is a product of value density times required computation time. If two or more phases share the same value density and the same required computation time, then any of the phases may be chosen.

4.3.2.2. DASA/ND Scheduling Automaton

The DASA/ND²⁸ Scheduling Automaton embodies the simplifications to the DASA Scheduling Automaton that can be made when there are no dependency issues to consider. The derivation of this simplified automaton appears in Appendix B. For the sake of convenience, the resulting automaton is presented in this section.

As before, each scheduling decision is made based on the set of phases currently known to the automaton and designated as the set $\{p_0, p_1, p_2, \dots\}$.

DASA/ND Automaton State Components. The state components associated with the DASA/ND scheduling automaton are presented in Figure 4-4. Since the algorithm-specific state components of the DASA Scheduling Automaton are all used to handle resources, they have been omitted in the DASA/ND Scheduling Automaton, leaving only the General State Components found in every scheduling automaton. (See Section 2.3.2.9.)

²⁸DASA/ND stands for DASA/No Dependencies.

General State Components:

- ExecMode: PHASE $\rightarrow$ MODE (MODE is either 'normal' or 'abort')
- ExecClock: PHASE $\rightarrow$ VIRTUAL-TIME
- AbortClock: PHASE $\rightarrow$ VIRTUAL-TIME
- ResumeTime: PHASE $\rightarrow$ TIMESTAMP
- Value: PHASE $\rightarrow$ (TIMESTAMP $\rightarrow$ VALUE) (initially Value(t) = 0)
- Total: VALUE (initially '0')
- RunningPhase: PHASE (initially 'nullphase')
- PhaseElect: MODE $\times$ PHASE (initially '<normal, nullphase>')
- PhaseList: list of PHASE (initially 'o')

Algorithm-Specific State Components:

- None

Figure 4-4: State Components of DASA/ND Scheduling Automaton

Operations Accepted by DASA/ND Automaton. The operations recognized by the DASA/ND Scheduling Automaton and their preconditions and postconditions are shown in Figure 4-5.

Once again, these are simpler than those shown previously for the DASA Scheduling Automaton in Section 3.2.1.2. In fact, largely because the automaton does not have to handle dependencies and aborts, this set of operation specifications is identical to that shown in the previous section for the LBESA Scheduling Automaton. Nonetheless, the two automata are not identical since their 'SelectPhase()' functions differ significantly.

'SelectPhase' Function for DASA/ND Automaton. Figure 4-6 shows the definition of the 'SelectPhase()' function for the DASA/ND Scheduling Automaton.

This definition is structurally similar to the definition for 'SelectPhase()' found in the LBESA Scheduling Automaton. But, although many of the functions are identical, there are some critical differences.

The most noticeable difference is the absence of the subsidiary function $P_{lastDL}()$, which locates the phase with the latest deadline. A less noticeable difference is invocation of $P_{leastPV}()$, rather than $P_{lastDL}()$, in the definition of $P_{scheduled}()$. In fact, it is $P_{scheduled}()$ that orders the phases as they are added to a tentative schedule for both the LBESA and the DASA/ND Scheduling Automata. Since the DASA/ND Scheduling Automaton adds phases to the schedule in order of decreasing potential value density, it has no need for $P_{lastDL}()$.

• t_{event} *request-phase*($v, t_{expected}$) P :

preconditions:

true <No preconditions here so that interrupts and other new phases can occur at any time>

postconditions:

if (RunningPhase = p) then

 if (ExecMode(p) = normal) then

 Total' = Total + Value(p)(t_{event})

 else

 ;no value for aborted phase

 ;release the resources acquired during the phase

 ;involves no action for this simplified automaton

Value'(p) = v

ExecClock'(p) = $t_{expected}$

AbortClock'(p) = 0

ExecMode'(p) = normal

;note that p is not resource-waiting

;make sure p is part of the list of phases, if necessary

if ($t_{expected} > 0$) then

 PhaseList' = PhaseList $\cup$ { p }

else

 PhaseList' = PhaseList - { p }

PhaseElect' = SelectPhase(PhaseList')

if (p = RunningPhase) then

 ;give up processor until next 'resume-phase'

 RunningPhase = nullphase

else

 ;happened under interrupt—leave 'RunningPhase' alone

• t_{event} *preempt-phase*(p) S :

preconditions:

(RunningPhase = p) $\wedge$ (RunningPhase $\neq$ nullphase)

$\wedge$ (RunningPhase $\neq$ Phase(PhaseElect))

postconditions:

if (ExecMode(p) = normal) then

 ExecClock'(p) = ExecClock(p) - (t_{event} - ResumeTime(p))

else

 AbortClock'(p) = AbortClock(p) - (t_{event} - ResumeTime(p))

;note p is not resource-waiting

RunningPhase' = nullphase

• t_{event} *resume-phase*(p) S :

preconditions:

(RunningPhase = nullphase) $\wedge$ (Phase(PhaseElect) = p)

$\wedge$ (Phase(PhaseElect) $\neq$ nullphase) $\wedge$ (Mode(PhaseElect) = normal)

postconditions:

ResumeTime'(p) = t_{event}

RunningPhase' = Phase(PhaseElect)

Figure 4-5: Operations Accepted by DASAND Scheduling Automaton

$SelectPhase(P) =$
 $\quad pickone(mustfinishby(DL_{first}(pmplist), P_{scheduled}(P))),$
 $\quad \text{where}$
 $\quad pmplist = tobescheduled(P_{scheduled}(P))$

$pickone(PMP) =$
 $\quad \langle normal, nullphase \rangle, \quad \text{if } PMP = \emptyset$
 $\quad \langle normal, p \rangle \mid \langle normal, p \rangle \in PMP, \quad \text{otherwise}$

$DL_{first}(PMP) =$
 $\quad \infty, \quad \text{if } PMP = \emptyset$
 $\quad Deadline(p) \mid (\langle normal, p \rangle \in PMP)$
 $\quad \wedge (\forall q)(\langle normal, q \rangle \in PMP \rightarrow Deadline(q) \geq Deadline(p)), \quad \text{otherwise}$

$P_{scheduled}(P) =$
 $\quad \emptyset, \quad \text{if } P = \emptyset$
 $\quad P_{feasible}(P_{scheduled}(P - \{p\}) \cup \{p\}), \quad \text{if } p \in P_{leastPV}(P)$

$P_{feasible}(P) = \emptyset, \quad \text{if } P = \emptyset$
 $\quad P, \quad \text{if } feasible(P)$
 $\quad P_{feasible}(P - \{p\}), \text{ where } p \in P_{leastPV}(P), \quad \text{otherwise}$

$P_{leastPV}(P) = \emptyset, \quad \text{if } P = \emptyset$
 $\quad \{p \mid (p \in P) \wedge (\forall q)(q \in P \rightarrow ((PVD(p) \leq PVD(q))$
 $\quad \wedge (PVD(p) = PVD(q) \rightarrow ExecClock(p) \leq ExecClock(q))))\}, \quad \text{otherwise}$

$tobescheduled(P) = \{ \langle normal, p \rangle \mid p \in P \}$

$mustcompleteby(t, P) =$
 $\quad \emptyset, \quad \text{if } t < t_{event}$
 $\quad \{p \mid [p \in P \wedge Deadline(p) \leq t]\}, \quad \text{otherwise}$

$mustfinishby(t, P) =$
 $\quad \emptyset, \quad \text{if } P = \emptyset \vee t < t_{event}$
 $\quad \vee mustcompleteby(t, P) = \emptyset$
 $\quad \{ \langle normal, p \rangle \mid p \in mustcompleteby(t, P) \}, \quad \text{otherwise}$

$feasible(P) = true, \text{ iff } (\forall t) \{ (t \geq t_{event}) \rightarrow timerequiredby(mustfinishby(t, P)) \leq (t - t_{event}) \}$

$timerequiredby(PMP) =$
 $\quad 0, \quad \text{if } PMP = \emptyset$
 $\quad ExecClock(p) + timerequiredby(PMP - \{ \langle normal, p \rangle \}), \text{ if } \langle normal, p \rangle \in PMP$

$PVD(p) = \frac{Val(p)}{ExecClock(p)}$

Figure 4-6: Functional Form of DASA/ND Algorithm

As was mentioned in the previous section, this difference in schedule construction may allow the DASA/ND Scheduling Automaton to accumulate a higher value for an application than the LBESA Scheduling Automaton. This will be illustrated by the proof in Section 4.3.2.4.

4.3.2.3. Proof: If No Overloads, c{DASA} and LBESA Are Equivalent

The introduction of the two scheduling automata in the previous sections has set the stage for the proofs in this section and the next. The proof that follows demonstrates that if there are no overloads, then both automata will accumulate the same value for the application — that is, both algorithms will make the same scheduling decisions. In fact, both automata will accept the same sequence of events as the scheduling automaton embodying a deadline scheduler. As stated earlier, a deadline-ordered schedule is known to be optimal for a uniprocessor when there are no overloads.

Theorem 2: Consider (1) a set of independent activities each comprising a single computational phase that is characterized by a simple time-value function — a step function with a positive value before a designated critical time and a value of zero after that time (that is, each phase has a hard deadline) — where (2) there are sufficient processor cycles to allow all of the phases to meet their deadlines. Given two automata, one designated DASA that accepts histories corresponding to schedules generated by the DASA Scheduling with Dependencies Algorithm and one designated LBESA that accepts histories corresponding to schedules generated by Locke's Best Effort Scheduling Algorithm, show that whenever (1) and (2) hold, every history that is accepted by LBESA is also accepted by DASA that has equal value. Thus both automata yield equal value for each such history.

Proof. For the sake of simplicity, since LBESA cannot handle dependencies among phases, this proof will be carried out by comparing the LBESA automaton with the DASA/ND automaton — a simplified version of the DASA Scheduling Automaton that contains no dependency considerations. The DASA/ND automaton is defined in Section 4.3.2.2. Furthermore, the only histories being examined by the automata are histories that do not involve overload situations.

Proof by induction.

Basis. Show that (1) if LBESA accepts the first event in a history, DASA/ND will also accept it, (2) *RunningPhase*, *PhaseList*, *PhaseElect*, and *Total* are the same for both automata, and (3) *Value*, *ExecClock*, *ExecMode*, and *ResumeTime* are the same for each active phase in both automata.

Initially,

```
RunningPhase=nullphase
PhaseList=0
PhaseElect=<normal,nullphase>
Total=0
```

As a result, the only event whose precondition for LBESA may be satisfied is 'request-phase.' Therefore, the first event in any history that LBESA will accept must be a 'request-phase.'

In that case, let the first event in the history be:

$t_{event1} \quad request-phase(v, t_{expected1}) \quad p_1$

LBESA accepts this event — its precondition for accepting it is *true* — and as part of its postconditions, it sets:

$Total' = 0$
 $Value'(p_1) = v$
 $ExecClock'(p_1) = t_{expected1}$
 $ExecMode'(p_1) = normal$
 $PhaseList' = \{p_1\}$
 $PhaseElect' = SelectPhase(PhaseList)$

 $= \langle normal, p_1 \rangle, \quad \text{if } feasible(\{p_1\})$
 $\quad \langle normal, nullphase \rangle, \quad \text{otherwise}$

DASAND also accepts this event — its precondition is also *true* — and, the state component changes induced by the postconditions for the event include those made by LBESA.

Therefore, DASAND accepts the first event in any history accepted by LBESA. Furthermore, *RunningPhase*, *PhaseList*, *PhaseElect*, and *Total* are identical in both automata after accepting this event. And finally, *Value*, *ExecClock*, and *ExecMode* are the same in both automata after accepting the event for the only currently active phase, p_1 , while *ResumeTime* is not yet defined for any phase in either automata — and so is trivially the same in both.

Inductive Step. Given that DASAND has accepted the first n events in a history that LBESA accepts; that *RunningPhase*, *PhaseList*, *PhaseElect*, and *Total* are the same for both automata after accepting those events; and that *Value*, *ExecClock*, *ExecMode*, and *ResumeTime* are the same in both automata for each active phase after accepting those events; show that DASAND will also accept the $n+1^{st}$ event in the history if LBESA accepts it, that *RunningPhase*, *PhaseList*, *PhaseElect*, and *Total* will be the same in each automaton after that event is accepted, and that *Value*, *ExecClock*, *ExecMode*, and *ResumeTime* will be the same in each automaton for each active phase after the $n+1^{st}$ event is accepted.

LBESA may accept any event for which the precondition is satisfied. In this case, it may accept an appropriate:

- 'preempt-phase'
- 'resume-phase'
- 'request-phase'

The precondition for accepting each of these events is the same in both automata. The preconditions depend only on the values of *RunningPhase*, *PhaseElect*, and the parameter p . Hence if LBESA accepts the $n+1^{st}$ event, DASAND will also accept it since, by inductive hypothesis, it has the same values for the relevant state components, the same parameter values, and the same precondition as LBESA.

Next, it should be demonstrated that *RunningPhase*, *PhaseList*, *PhaseElect*, and *Total* are the same for

both automata after the $n+1^{st}$ event is accepted, and that *Value*, *ExecClock*, *ExecMode*, and *ResumeTime* are the same for both automata for each active phase after that event is accepted.

Consider each of the three possible events:

1. 'preempt-phase' — in both automata, *RunningPhase'* is set to *nullphase* while *PhaseElect* remains unchanged, and therefore equal. Also in both automata, *ExecClock'(p)* is conditionally assigned a new value. In both automata, the condition — *ExecMode(p)=normal* — is identical, *p* is the same in both since it is part of the $n+1^{st}$ event, and by inductive assumption *ExecMode(p)* is the same in both automata. Consequently, either both or neither of the automata will update *ExecClock'(p)*. Finally, note that the formula used to update *ExecClock'(p)* is the same in both automata, *ExecClock(p)* and *ResumeTime(p)* are the same in both automata by inductive assumption, and t_{event} is the same in both since it is part of the $n+1^{st}$ event and so is independent of the state of the automata. Therefore, *ExecClock'(p)* will be the same in both automata if it is updated.
2. 'resume-phase' — in both automata, *RunningPhase* is set to *PhaseElect*, which has the same value in both automata after accepting the first *n* events, while *PhaseElect* remains unchanged, and therefore equal, in both automata. Also, *ResumeTime'(p)* is set to t_{event} . This assignment results in the same state for *ResumeTime'(p)* in both automata since *ResumeTime* was the same in both automata for all active phases after the first *n* events had been accepted and t_{event} is the same for both automata because it is part of the $n+1^{st}$ event and so is independent of the states of the automata.
3. 'request-phase' —
 - *RunningPhase*: in both automata, *RunningPhase'* may conditionally be set to *nullphase*; in each automaton, the condition under which this is done — $p=RunningPhase$ — is the same, *RunningPhase* is the same by inductive hypothesis, and *p* is the same since it is part of the $n+1^{st}$ event and has a value that is independent of the state of the automaton.
 - *PhaseList*: in both automata, *PhaseList'* will conditionally be set to either $PhaseList \cup \{p\}$ or $PhaseList - \{p\}$; in each automaton, the condition under which this is done — $t_{expected} > 0$ — is the same, *PhaseList* is the same by inductive hypothesis, and *p* and $t_{expected}$ are the same since they are part of the $n+1^{st}$ event and have values that are independent of the state of the automaton.
 - *PhaseElect*: in both automata, *PhaseElect'* is set to *SelectPhase(PhaseList')*. As argued in the previous bullet, *PhaseList'* is the same in both automata. Now consider the function *SelectPhase()* for each automaton. Most of the subordinate functions involved in the definition of *SelectPhase()* are identical in both automata. In fact the only subordinate function that differs is $P_{scheduled}()$ — although the form is the same in both, the specific ordering of recursive functional evaluations is different in the two automaton definitions.

It is given that there are sufficient processor cycles available to allow all of the phases to meet all of their deadlines. In terms of the mathematical formulation of these automata, this is equivalent to saying that $(\forall P) feasible(P)=true$ — that is, it is feasible to schedule all of the known phases at any given time.²⁹ In that case, for both automata the definition of $P_{feasible}()$ can be simplified from

$$P_{feasible}(P) = \begin{cases} \emptyset, & \text{if } P = \emptyset \\ P, & \text{if } feasible(P) \\ P_{feasible}(P - \{p\}), \text{ where } p \in P_{leastP1}(P), & \text{otherwise} \end{cases}$$

²⁹This property is maintained through an entire history because both automata accept deadline-ordered histories if there are no overload conditions, and a deadline-ordered schedule will be guaranteed to meet all of the deadlines if there are sufficient processing cycles available.

to

$$P_{feasible}(P) = \begin{cases} \emptyset, & \text{if } P = \emptyset \\ P, & \text{if } feasible(P) \end{cases}$$

and, finally, to

$$P_{feasible}(P) = P$$

Since $P_{feasible}()$ acts as an identity function, the definition of $P_{scheduled}()$ can also be simplified from

$$P_{scheduled}(P) = \begin{cases} \emptyset, & \text{if } P = \emptyset \\ P_{feasible}(P_{scheduled}(P - \{p\}) \cup \{p\}), & \text{if } p \in P_x(P) \end{cases}$$

to

$$P_{scheduled}(P) = \begin{cases} \emptyset, & \text{if } P = \emptyset \\ P_{scheduled}(P - \{p\}) \cup \{p\}, & \text{if } p \in P_x(P) \end{cases}$$

which is equivalent to

$$P_{scheduled}(P) = P$$

Of course, the definitions of *SelectPhase()* for both automata are now exactly the same. The evaluation of *SelectPhase(PhaseList')* depends on the values of *PhaseList'*, *ExecClock'*, and *Value'*, all of which are shown to be the same for both automata after the $n+1_{st}$ event in the history is accepted. The value also depends on t_{event} which is the same for both automata since it is part of the $n+1_{st}$ event and is therefore independent of the state of the automata. Consequently, *PhaseElect'* will be the same for both automata.

- *Total*: in both automata, if *RunningPhase*= p and *ExecMode*(p)=*normal*, *Total'* will be set equal to $Total + Value(p)(t_{event})$; otherwise, *Total'* will remain unchanged by the $n+1_{st}$ event. Since by inductive hypothesis *RunningPhase* and *ExecMode* are the same in both automata, and since p is the same in both automata since it is part of the $n+1_{st}$ event and consequently has a value that is independent of the state of the automata, the condition under which *Total'* will be updated by the $n+1_{st}$ event is the same in both automata. Also, since *Total* and *Value* are the same in both automata by inductive hypothesis, and p and t_{event} are both part of the $n+1_{st}$ event, the computed value assigned to *Total'* is identical in both automata.
- *Value*: in both automata, *Value'*(p) is unconditionally set to v ; in each automaton, p and v are the same since they are part of the $n+1_{st}$ event and have values that are independent of the state of the automaton. Since *Value* had been the same in both automata after n events were accepted and since both automata set *Value'*(p)= v , *Value'* is the same in both.
- *ExecClock*: in both automata, *ExecClock'*(p) is unconditionally set to $t_{expected}$; in each automaton, p and $t_{expected}$ are the same since they are part of the $n+1_{st}$ event and have values that are independent of the state of the automaton. Since *ExecClock* had been the same in both automata after n events were accepted and since both automata set *ExecClock'*(p)= $t_{expected}$, *ExecClock'* is the same in both.
- *ExecMode*: in both automata, *ExecMode'*(p) is unconditionally set to *normal*; in each automaton, p is the same since it is part of the $n+1_{st}$ event and has a value that is independent of the state of the automaton. Since *ExecMode* had been the same in both automata after n events were accepted and since both automata set *ExecMode'*(p)=*normal*, *ExecMode'* is the same in both.
- *ResumeTime*: in both automata, *ResumeTime* remains unchanged.

Thus, if LBESA accepts any of these event types as the $n+1^{st}$ event in a history, so will DASAND, and the significant state components of each automaton will be the same at that point.

Therefore, by induction, DASAND, and hence DASA itself, accepts any history accepted by LBESA under the conditions outlined in the theorem statement. Furthermore, because the state component *Total* will be the same in both automata after accepting a history, both will yield the same value for any history that they accept.

EndOfProof

4.3.2.4. Proof: With Overloads, DASA May Exceed LBESA

The preceding section showed that the DASA Scheduling Automaton performed well when there were no overloads. In this section, it is shown that, when there are overloads, the DASA Scheduling Automaton may accept histories that yield higher values for the application than any history that may be accepted by the LBESA Scheduling Automaton. The reason for this has been mentioned previously in Sections 4.3.2.1 and 4.3.2.2: although both algorithms use similar value density metrics, they construct schedules in different ways, potentially resulting in situations where LBESA sheds some phases unnecessarily.

Once again, for the sake of simplicity, since no dependencies are involved, the DASAND Scheduling Automaton, rather than the DASA Scheduling Automaton, is used in the following proof. The result, however, applies to the DASA Scheduling Automaton as well.

Theorem 3: Consider (1) a set of independent activities each comprising a single computational phase that is characterized by a simple time-value function — a step function with a positive value before a designated critical time and a value of zero after that time (that is, each phase has a hard deadline) — where (2) there are insufficient processor cycles to allow all of the phases to meet their deadlines. Given two automata, one designated DASAND that accepts histories corresponding to schedules generated by the DASA Scheduling with Dependencies Algorithm and one designated LBESA that accepts histories corresponding to schedules generated by Locke's Best Effort Scheduling Algorithm, show that there are situations where (1) and (2) hold and DASAND will accept a history with a greater value than any history LBESA will accept involving the same phases and the same scheduling parameters (time-value functions and required computation times).

Proof. This proof is carried out by constructing an example.

Intuitively, LBESA constructs a complete schedule by considering each phase in order of its deadline, nearest deadline first. As each phase is considered, an estimate is performed to determine whether there is an overload situation. In that case, it discards the phases with the lowest value densities until a feasible schedule is obtained. In the process, it may discard some phases unnecessarily. DASAND also constructs a schedule from scratch; however, it begins with the phase having the greatest value density and considers subsequent phases in decreasing order of value density. Each phase is included in the schedule in order of its deadline if the schedule — including that phase — is feasible. Since this approach includes as many

high value density phases as possible and only discards those phases that cannot be added to the schedule, rather than those that have lower value densities than one that must be discarded. It avoids the problem that LBESA encounters.

An example of DASA/ND accepting a history with a greater value than LBESA will accept can be constructed using phases with the following parameters:

Phase	Deadline	Required Computation Time	Value
p1	$2t_a$	$t_a + 1$	v1
p2	$2t_a$	t_a	v2
p3	$2t_a - 1$	$t_a - 1$	v3

Let $t_a \geq 1$ and $v1, v2, v3 > 0$. Also, let $v1/(t_a+1) > v2/t_a > v3/(t_a-1)$ indicate the initial relationship of the value densities of the three phases, p1, p2, and p3, respectively. Now consider the following history, H_f :

$t_1 = 0^-$	request-phase(step(v1, $2t_a$), t_a+1)	p1
$t_2 = 0^-$	request-phase(step(v2, $2t_a$), t_a)	p2
$t_3 = 0$	request-phase(step(v3, $2t_a-1$), t_a-1)	p3
$t_4 = 0^+$	resume-phase(p3)	S
$t_5 = t_a-1$	request-phase(step(0, ∞), 0)	p3
$t_6 = (t_a-1)^+$	resume-phase(p1)	S
$t_7 = 2t_a$	request-phase(step(0, ∞), 0)	p1

This history is accepted by the DASA/ND automaton and has a value of $v1+v3$, but is not accepted by the LBESA automaton. In fact, the only histories that LBESA accepts with only these three phases and the same scheduling parameters have value v1 or less. The following sections demonstrate each of these facts in turn.

DASA/ND Automaton Accepts History H_f . By following the DASA/ND automaton through the state changes accompanying the acceptance of each individual event in the history, this section will demonstrate that history H_f is accepted by DASA/ND. For reference, the DASA/ND automaton was defined in Section 4.3.2.2.

According to the automaton definition, initially:

$Total=0$
 $RunningPhase=nullphase$
 $PhaseElect=<normal,nullphase>$
 $PhaseList=\emptyset$

The following labeled steps demonstrate the acceptance of each event in history H_f and detail the changes in state component values that accompany each event.

Event 1: $t_1 = 0^-$ request-phase(step(v1, 2t _a), t _a +1)	p1
--	----

event parameters:

$$\begin{aligned}
 t_{event} &= t_1 = 0^- \\
 v &= \text{step}(v1, 2t_a) \\
 t_{expected} &= t_a + 1 \\
 p &= p1
 \end{aligned}$$

precondition: *true*

(so the event is accepted)

postconditions:

$$\begin{aligned}
 \text{Value}'(p1) &= \text{step}(v1, 2t_a) \\
 \text{ExecClock}'(p1) &= t_a + 1 \\
 \text{AbortClock}'(p1) &= 0 \\
 \text{ExecMode}'(p1) &= \text{normal} \\
 \text{PhaseList}' &= \emptyset \cup \{p1\} = \{p1\} \\
 \text{PhaseElect}' &= \text{SelectPhase}(\{p1\})
 \end{aligned}$$

(since $t_{expected} > 0$)

Event 2: $t_2 = 0^-$ request-phase(step(v2, 2t _a), t _a)	p2
--	----

event parameters:

$$\begin{aligned}
 t_{event} &= t_2 = 0^- \\
 v &= \text{step}(v2, 2t_a) \\
 t_{expected} &= t_a \\
 p &= p2
 \end{aligned}$$

precondition: *true*

(so the event is accepted)

postconditions:

$$\begin{aligned}
 \text{Value}'(p2) &= \text{step}(v2, 2t_a) \\
 \text{ExecClock}'(p2) &= t_a \\
 \text{AbortClock}'(p2) &= 0 \\
 \text{ExecMode}'(p2) &= \text{normal} \\
 \text{PhaseList}' &= \{p1\} \cup \{p2\} = \{p1, p2\} \\
 \text{PhaseElect}' &= \text{SelectPhase}(\{p1, p2\})
 \end{aligned}$$

(since $t_{expected} > 0$)

Event 3: $t_3 = 0$ request-phase(step(v3, 2t _a -1), t _a -1)	p3
--	----

event parameters:

$$\begin{aligned}
 t_{event} &= t_3 = 0 \\
 v &= \text{step}(v3, 2t_a - 1) \\
 t_{expected} &= t_a - 1 \\
 p &= p3
 \end{aligned}$$

precondition: *true*

(so the event is accepted)

postconditions:

$Value'(p3) = step(v3, 2t_a - 1)$
 $ExecClock'(p3) = t_a - 1$
 $AbortClock'(p3) = 0$
 $ExecMode'(p3) = normal$
 $PhaseList' = \{p1, p2\} \cup \{p3\} = \{p1, p2, p3\}$
 $PhaseElect' = SelectPhase(\{p1, p2, p3\})$

(since $t_{expected} > 0$)

Evaluating $SelectPhase(\{p1, p2, p3\}) \dots$

$SelectPhase(\{p1, p2, p3\}) =$
 $pickone(mustfinishby(DL_{first}(pmplist), P_{scheduled}(\{p1, p2, p3\}))),$
 where
 $pmplist = tobescheduled(P_{scheduled}(\{p1, p2, p3\}))$

$P_{scheduled}(\{p1, p2, p3\}) =$
 $P_{feasible}(P_{scheduled}(\{p1, p2\}) \cup \{p3\})$

(since $P_{leastPV}(\{p1, p2, p3\}) = \{p3\}$)

$P_{scheduled}(\{p1, p2\}) =$
 $P_{feasible}(P_{scheduled}(\{p1\}) \cup \{p2\})$

(since $P_{leastPV}(\{p1, p2\}) = \{p2\}$)

$P_{scheduled}(\{p1\})$
 $= P_{feasible}(P_{scheduled}(0) \cup \{p1\})$

(since $P_{leastPV}(\{p1\}) = \{p1\}$)

$= P_{feasible}(0 \cup \{p1\})$
 $= P_{feasible}(\{p1\})$
 $= \{p1\}, \text{ if feasible}(\{p1\})$

$feasible(\{p1\}) = true,$
 iff $(\forall t)[(t \geq t_{event}) \rightarrow timerequiredby(mustfinishby(t, \{p1\})) \leq (t - t_{event})]$

For $t \geq t_{event} \dots$

$mustfinishby(t, \{p1\}) =$
 $0,$ if $mustcompleteby(t, \{p1\}) = 0$
 $\{ \langle normal, p \rangle \mid p \in mustcompleteby(t, \{p1\}) \},$
 otherwise

$mustcompleteby(t, \{p1\})$
 $= \{p \mid [p \in \{p1\} \wedge Deadline(p) \leq t]\}$
 $= 0,$ if $t < Deadline(p1) = 2t_a$
 $\{p1\},$ if $t \geq Deadline(p1) = 2t_a$

Therefore ...

$$\begin{aligned}
\text{mustfinishby}(t, \{p1\}) &= 0, && \text{if } t < 2t_a \\
&\{ \langle \text{normal}, p \rangle \mid p \in \{p1\} \}, && \text{otherwise} \\
&= 0, && \text{if } t < 2t_a \\
&\{ \langle \text{normal}, p1 \rangle \}, && \text{otherwise } (t \geq 2t_a) \\
\text{timerequiredby}(\text{mustfinishby}(t, \{p1\})) &= \\
&= \text{timerequiredby}(0), && \text{if } t < 2t_a \\
&\text{timerequiredby}(\{ \langle \text{normal}, p1 \rangle \}), && \text{if } t \geq 2t_a \\
&= 0, && \text{if } t < 2t_a \\
&\text{ExecClock}(p1) = t_a + 1, && \text{if } t \geq 2t_a
\end{aligned}$$

Notice that for $t \geq t_{\text{event}} = 0$, when $t < 2t_a$...

$$\text{timerequiredby}(\text{mustfinishby}(t, \{p1\})) = 0 \leq (t - t_{\text{event}}), \quad \text{as required for feasibility}$$

And when $t \geq 2t_a$...

$$\text{timerequiredby}(\text{mustfinishby}(t, \{p1\})) = t_a + 1 \leq 2t_a \leq (t - t_{\text{event}}), \quad \text{as required for feasibility}$$

Therefore ...

$$\text{feasible}(\{p1\}) = \text{true} \rightarrow P_{\text{scheduled}}(\{p1\}) = \{p1\}$$

Continuing ...

$$\begin{aligned}
P_{\text{scheduled}}(\{p1, p2\}) &= P_{\text{feasible}}(P_{\text{scheduled}}(\{p1\}) \cup \{p2\}) \\
&= P_{\text{feasible}}(\{p1\} \cup \{p2\}) \\
&= P_{\text{feasible}}(\{p1, p2\})
\end{aligned}$$

To evaluate $P_{\text{feasible}}(\{p1, p2\})$...

$$\begin{aligned}
\text{feasible}(\{p1, p2\}) &= \text{true, iff } (\forall t)[(t \geq t_{\text{event}}) \\
&\rightarrow \text{timerequiredby}(\text{mustfinishby}(t, \{p1, p2\})) \leq (t - t_{\text{event}})]
\end{aligned}$$

For $t \geq t_{\text{event}}$...

$$\begin{aligned}
\text{mustfinishby}(t, \{p1, p2\}) &= \\
&0, && \text{if } \text{mustcompleteby}(t, \{p1, p2\}) = 0 \\
&\{ \langle \text{normal}, p \rangle \mid p \in \text{mustcompleteby}(t, \{p1, p2\}) \}, && \text{otherwise} \\
\text{mustcompleteby}(t, \{p1, p2\}) &= \{ p \mid [p \in \{p1, p2\} \wedge \text{Deadline}(p) \leq t] \} \\
&= 0, && \text{if } t < \text{Deadline}(p1) = \text{Deadline}(p2) = 2t_a \\
&\{p1, p2\}, && \text{if } t \geq \text{Deadline}(p1) = \text{Deadline}(p2) = 2t_a
\end{aligned}$$

Therefore ...

$$\begin{aligned}
 \text{mustfinishby}(t, \{p1, p2\}) &= \emptyset, & \text{if } t < 2t_a \\
 &= \{ \langle \text{normal}, p \rangle \mid p \in \{p1, p2\} \}, & \text{otherwise} \\
 &= \emptyset, & \text{if } t < 2t_a \\
 &= \{ \langle \text{normal}, p1 \rangle, \langle \text{normal}, p2 \rangle \}, & \text{otherwise } (t \geq 2t_a) \\
 \text{timerequiredby}(\text{mustfinishby}(t, \{p1, p2\})) &= \\
 &= \text{timerequiredby}(\emptyset), & \text{if } t < 2t_a \\
 &= \text{timerequiredby}(\{ \langle \text{normal}, p1 \rangle, \langle \text{normal}, p2 \rangle \}), & \text{if } t \geq 2t_a \\
 &= 0, & \text{if } t < 2t_a \\
 &= \text{ExecClock}(p1) + \text{ExecClock}(p2) = 2t_a + 1, & \text{if } t \geq 2t_a
 \end{aligned}$$

Notice that for $t = 2t_a$...

$$\text{timerequiredby}(\text{mustfinishby}(t, \{p1, p2\})) = 2t_a + 1 > 2t_a = (t - t_{\text{event}})$$

This violates the requirement for feasibility, therefore ...

$$\begin{aligned}
 \text{feasible}(\{p1, p2\}) &= \text{false} \\
 &\rightarrow P_{\text{feasible}}(\{p1, p2\}) = P_{\text{feasible}}(\{p1\}) & (\text{since } P_{\text{feasible}}(\{p1, p2\}) = \{p2\}) \\
 &= \{p1\} & (\text{as shown above}) \\
 &\rightarrow P_{\text{scheduled}}(\{p1, p2\}) = \{p1\} & (\text{since } P_{\text{scheduled}}(\{p1, p2\}) = P_{\text{feasible}}(\{p1, p2\}))
 \end{aligned}$$

Continuing ...

$$\begin{aligned}
 P_{\text{scheduled}}(\{p1, p2, p3\}) &= P_{\text{feasible}}(P_{\text{scheduled}}(\{p1, p2\}) \cup \{p3\}) & (\text{as shown above}) \\
 &= P_{\text{feasible}}(\{p1\} \cup \{p3\}) \\
 &= P_{\text{feasible}}(\{p1, p3\})
 \end{aligned}$$

To evaluate $P_{\text{feasible}}(\{p1, p3\})$...

$$\begin{aligned}
 \text{feasible}(\{p1, p3\}) &= \text{true, iff } (\forall t) \{ (t \geq t_{\text{event}}) \\
 &\rightarrow \text{timerequiredby}(\text{mustfinishby}(t, \{p1, p3\})) \leq (t - t_{\text{event}}) \}
 \end{aligned}$$

For $t \geq t_{\text{event}}$...

$$\begin{aligned} \text{mustfinishby}(t, \{p1, p3\}) = & \\ & 0, \quad \text{if } \text{mustcompleteby}(t, \{p1, p3\}) = 0 \\ & \{ \langle \text{normal}, p \rangle \mid p \in \text{mustcompleteby}(t, \{p1, p3\}) \}, \quad \text{otherwise} \end{aligned}$$

$$\begin{aligned} \text{mustcompleteby}(t, \{p1, p3\}) & \\ & = \{ p \mid [p \in \{p1, p3\} \wedge \text{Deadline}(p) \leq t] \} \\ & = 0, \quad \text{if } t < \text{Deadline}(p3) = 2t_a - 1 \\ & \quad \{p3\}, \quad \text{if } \text{Deadline}(p3) = 2t_a - 1 \leq t < \text{Deadline}(p1) = 2t_a \\ & \quad \{p1, p3\}, \quad \text{if } t \geq \text{Deadline}(p1) = 2t_a \end{aligned}$$

Therefore ...

$$\begin{aligned} \text{mustfinishby}(t, \{p1, p3\}) & \\ & = 0, \quad \text{if } t < 2t_a - 1 \\ & \quad \{ \langle \text{normal}, p \rangle \mid p \in \{p3\} \}, \quad \text{if } 2t_a - 1 \leq t < 2t_a \\ & \quad \{ \langle \text{normal}, p \rangle \mid p \in \{p1, p3\} \}, \quad \text{if } 2t_a \leq t \\ & = 0, \quad \text{if } t < 2t_a - 1 \\ & \quad \{ \langle \text{normal}, p3 \rangle \}, \quad \text{if } 2t_a - 1 \leq t < 2t_a \\ & \quad \{ \langle \text{normal}, p1 \rangle, \langle \text{normal}, p3 \rangle \}, \quad \text{if } 2t_a \leq t \\ \text{timerequiredby}(\text{mustfinishby}(t, \{p1, p3\})) & \\ & = \text{timerequiredby}(0), \quad \text{if } t < 2t_a - 1 \\ & \quad \text{timerequiredby}(\{ \langle \text{normal}, p3 \rangle \}), \quad \text{if } 2t_a - 1 \leq t < 2t_a \\ & \quad \text{timerequiredby}(\{ \langle \text{normal}, p1 \rangle, \langle \text{normal}, p3 \rangle \}), \quad \text{if } 2t_a \leq t \\ & = 0, \quad \text{if } t < 2t_a - 1 \\ & \quad \text{ExecClock}(p3) = t_a - 1, \quad \text{if } 2t_a - 1 \leq t < 2t_a \\ & \quad \text{ExecClock}(p1) + \text{ExecClock}(p3) = 2t_a, \quad \text{if } 2t_a \leq t \end{aligned}$$

Notice that for $t \geq t_{\text{event}} = 0$, when $t < 2t_a - 1$...

$$\text{timerequiredby}(\text{mustfinishby}(t, \{p1, p3\})) = 0 \leq (t - t_{\text{event}}), \quad \text{as required for feasibility}$$

When $2t_a - 1 \leq t < 2t_a$...

$$\text{timerequiredby}(\text{mustfinishby}(t, \{p1, p3\})) = t_a - 1 < 2t_a - 1 \leq (t - t_{\text{event}}), \quad \text{as required for feasibility}$$

And when $t \geq 2t_a$...

$$\text{timerequiredby}(\text{mustfinishby}(t, \{p1, p3\})) = 2t_a \leq (t - t_{\text{event}}), \quad \text{as required for feasibility}$$

Therefore ...

$$\begin{aligned} \text{feasible}(\{p1, p3\}) &= \text{true} \\ &\rightarrow P_{\text{feasible}}(\{p1, p3\}) = \{p1, p3\} \\ &\rightarrow P_{\text{scheduled}}(\{p1, p2, p3\}) = \{p1, p3\} \end{aligned}$$

So $P_{\text{scheduled}}(\{p1, p2, p3\})$, the set of phases that can feasibly be executed so that each will meet its deadline while contributing the maximum value to the system for the investment of a given amount of time, has now been determined. Next, the individual phase from this set that will be executed first must be determined . . .

$$\begin{aligned} pmplist &= \text{to_scheduled}(P_{\text{scheduled}}(\{p1, p2, p3\})) \\ &= \text{to_scheduled}(\{p1, p3\}) \\ &= \{ \langle \text{normal}, p1 \rangle, \langle \text{normal}, p3 \rangle \} \end{aligned}$$

$$\begin{aligned} DL_{\text{first}}(pmplist) &= DL_{\text{first}}(\{ \langle \text{normal}, p1 \rangle, \langle \text{normal}, p3 \rangle \}) \\ &= 2t_a - 1 \end{aligned}$$

$$\begin{aligned} \text{mustfinishby}(DL_{\text{first}}(pmplist), P_{\text{scheduled}}(\{p1, p2, p3\})) \\ &= \text{mustfinishby}(2t_a - 1, \{p1, p3\}) \\ &= \{ \langle \text{normal}, p \rangle \mid p \in \text{mustcompleteby}(2t_a - 1, \{p1, p3\}) \} \\ &\quad \text{(assuming } \text{mustcompleteby}(2t_a - 1, \{p1, p3\}) \neq \emptyset) \\ &= \{ \langle \text{normal}, p \rangle \mid p \in \{p1, p3\} \wedge \text{Deadline}(p) \leq 2t_a - 1 \} \\ &= \{ \langle \text{normal}, p \rangle \mid p \in \{p3\} \} \quad \text{(note that } \text{mustcompleteby}(2t_a - 1, \{p1, p3\}) \neq \emptyset) \\ &= \{ \langle \text{normal}, p3 \rangle \} \end{aligned}$$

Finally . . .

$$\begin{aligned} \text{PhaseElect}' &= \text{SelectPhase}(\{p1, p2, p3\}) \\ &= \text{pickone}(\text{mustfinishby}(DL_{\text{first}}(pmplist), P_{\text{scheduled}}(\{p1, p2, p3\}))) \\ &= \text{pickone}(\{ \langle \text{normal}, p3 \rangle \}) \\ &= \langle \text{normal}, p3 \rangle \end{aligned}$$

Event 4: $t_4 = 0^+$ resume-phase(p3)	S
---	----------

event parameters:

$$\begin{aligned} t_{\text{event}} &= t_4 = 0^+ \\ p &= p3 \end{aligned}$$

precondition:

$$\begin{aligned} &(\text{RunningPhase} = \text{nullphase}) \wedge (\text{Phase}(\text{PhaseElect}) = p3) \\ &\quad \wedge (\text{Phase}(\text{PhaseElect}) \neq \text{nullphase}) \wedge (\text{Mode}(\text{PhaseElect}) = \text{normal}) \end{aligned}$$

≡

$$\begin{aligned} &(\text{RunningPhase} = \text{nullphase}) \wedge (\text{Phase}(\langle \text{normal}, p3 \rangle) = p3) \\ &\quad \wedge (\text{Phase}(\langle \text{normal}, p3 \rangle) \neq \text{nullphase}) \wedge (\text{Mode}(\langle \text{normal}, p3 \rangle) = \text{normal}) \end{aligned}$$

≡

true

(so the event is accepted)

postconditions:

$$\begin{aligned} \text{ResumeTime}'(p3) &= 0^+ \\ \text{RunningPhase}' &= \text{Phase}(\text{PhaseElect}) = \text{Phase}(\langle \text{normal}, p3 \rangle) = p3 \end{aligned}$$

Event 5: $t_5 = t_a - 1$ request-phase(step(0, ∞), 0)	p3
---	----

event parameters:

$$\begin{aligned} t_{event} &= t_5 = t_a - 1 \\ v &= \text{step}(0, \infty) \\ t_{expected} &= 0 \\ p &= p3 \end{aligned}$$

precondition: *true*

(so the event is accepted)

postconditions:

$$\begin{aligned} Total' &= 0 + \text{Value}(p3)(t_a - 1) && (\text{since } \text{RunningPhase} = p3 \wedge \text{ExecMode}(p3) = \text{normal}) \\ &= \text{step}(v3, 2t_a - 1)(t_a - 1) \\ &= v3 \\ \text{Value}'(p3) &= \text{step}(0, \infty) \\ \text{ExecClock}'(p3) &= 0 \\ \text{AbortClock}'(p3) &= 0 \\ \text{ExecMode}'(p3) &= \text{normal} \\ \text{PhaseList}' &= \{p1, p2, p3\} - \{p3\} = \{p1, p2\} && (\text{since } t_{expected} = 0) \\ \text{PhaseElect}' &= \text{SelectPhase}(\{p1, p2\}) \\ \text{RunningPhase}' &= \text{nullphase} && (\text{since } p3 = \text{RunningPhase}) \end{aligned}$$

Evaluating $\text{SelectPhase}(\{p1, p2\}) \dots$

$$\begin{aligned} \text{SelectPhase}(\{p1, p2\}) &= \\ &\text{pickone}(\text{mustfinishby}(DL_{\text{first}}(pmplist), P_{\text{scheduled}}(\{p1, p2\}))), \\ &\text{where} \\ &pmplist = \text{tobescheduled}(P_{\text{scheduled}}(\{p1, p2\})) \end{aligned}$$

$$\begin{aligned} P_{\text{scheduled}}(\{p1, p2\}) &= \\ &P_{\text{feasible}}(P_{\text{scheduled}}(\{p1\}) \cup \{p2\}) && (\text{since } P_{\text{leastPV}}(\{p1, p2\}) = \{p2\}) \end{aligned}$$

$$\begin{aligned} P_{\text{scheduled}}(\{p1\}) &= \\ &= P_{\text{feasible}}(P_{\text{scheduled}}(\emptyset) \cup \{p1\}) && (\text{since } P_{\text{leastPV}}(\{p1\}) = \{p1\}) \\ &= P_{\text{feasible}}(\emptyset \cup \{p1\}) \\ &= P_{\text{feasible}}(\{p1\}) \\ &= \{p1\}, \text{if feasible}(\{p1\}) \end{aligned}$$

$$\begin{aligned} \text{feasible}(\{p1\}) &= \text{true}, \\ &\text{iff } (\forall t)[(t \geq t_{event}) \rightarrow \text{timerequiredby}(\text{mustfinishby}(t, \{p1\})) \leq (t - t_{event})] \end{aligned}$$

As before, for $t \geq t_{event} = t_a - 1 \dots$

$$\begin{aligned} \text{mustfinishby}(t, \{p1\}) = & \emptyset, & \text{if } \text{mustcompleteby}(t, \{p1\}) = \emptyset \\ & \{ \langle \text{normal}, p \rangle \mid p \in \text{mustcompleteby}(t, \{p1\}) \}, & \text{otherwise} \end{aligned}$$

$$\begin{aligned} \text{mustcompleteby}(t, \{p1\}) &= \{ p \mid [p \in \{p1\} \wedge \text{Deadline}(p) \leq t] \} \\ &= \emptyset, & \text{if } t < \text{Deadline}(p1) = 2t_a \\ & \{p1\}, & \text{if } t \geq \text{Deadline}(p1) = 2t_a \end{aligned}$$

Therefore ...

$$\begin{aligned} \text{mustfinishby}(t, \{p1\}) &= \emptyset, & \text{if } t < 2t_a \\ & \{ \langle \text{normal}, p \rangle \mid p \in \{p1\} \}, & \text{otherwise} \\ &= \emptyset, & \text{if } t < 2t_a \\ & \{ \langle \text{normal}, p1 \rangle \}, & \text{otherwise } (t \geq 2t_a) \\ \text{timerequiredby}(\text{mustfinishby}(t, \{p1\})) &= \text{timerequiredby}(\emptyset), & \text{if } t < 2t_a \\ & \text{timerequiredby}(\{ \langle \text{normal}, p1 \rangle \}), & \text{if } t \geq 2t_a \\ &= 0, & \text{if } t < 2t_a \\ & \text{ExecClock}(p1) = t_a + 1, & \text{if } t \geq 2t_a \end{aligned}$$

Notice that for $t \geq t_{\text{event}} = t_a - 1$, when $t < 2t_a$...

$$\text{timerequiredby}(\text{mustfinishby}(t, \{p1\})) = 0 \leq (t - t_{\text{event}}), \quad \text{as required for feasibility}$$

And when $t \geq 2t_a$...

$$\text{timerequiredby}(\text{mustfinishby}(t, \{p1\})) = t_a + 1 \leq (t - t_{\text{event}}), \quad \text{as required for feasibility}$$

Therefore ...

$$\text{feasible}(\{p1\}) = \text{true} \rightarrow P_{\text{scheduled}}(\{p1\}) = \{p1\}$$

Continuing ...

$$\begin{aligned} P_{\text{scheduled}}(\{p1, p2\}) &= P_{\text{feasible}}(P_{\text{scheduled}}(\{p1\}) \cup \{p2\}) & (\text{as shown above}) \\ &= P_{\text{feasible}}(\{p1\} \cup \{p2\}) \\ &= P_{\text{feasible}}(\{p1, p2\}) \end{aligned}$$

$$\begin{aligned} \text{feasible}(\{p1, p2\}) = \text{true, iff } (\forall t) [(t \geq t_{\text{event}}) \\ \rightarrow \text{timerequiredby}(\text{mustfinishby}(t, \{p1, p2\})) \leq (t - t_{\text{event}})] \end{aligned}$$

As before, for $t \geq t_{\text{event}}$...

$$\begin{aligned}
\text{mustfinishby}(t, \{p1, p2\}) &= \\
&\quad \emptyset, && \text{if } \text{mustcompleteby}(t, \{p1, p2\}) = 0 \\
&\quad \{ \langle \text{normal}, p \rangle \mid p \in \text{mustcompleteby}(t, \{p1, p2\}) \}, && \text{otherwise} \\
\text{mustcompleteby}(t, \{p1, p2\}) &= \\
&\quad \{ p \mid [p \in \{p1, p2\} \wedge \text{Deadline}(p) \leq t] \} \\
&\quad \emptyset, && \text{if } t < \text{Deadline}(p1) = \text{Deadline}(p2) = 2t_a \\
&\quad \{p1, p2\}, && \text{if } t \geq \text{Deadline}(p1) = \text{Deadline}(p2) = 2t_a
\end{aligned}$$

Therefore ...

$$\begin{aligned}
\text{mustfinishby}(t, \{p1, p2\}) &= \\
&\quad \emptyset, && \text{if } t < 2t_a \\
&\quad \{ \langle \text{normal}, p \rangle \mid p \in \{p1, p2\} \}, && \text{otherwise} \\
&\quad \emptyset, && \text{if } t < 2t_a \\
&\quad \{ \langle \text{normal}, p1 \rangle, \langle \text{normal}, p2 \rangle \}, && \text{otherwise } (t \geq 2t_a) \\
\text{timerequiredby}(\text{mustfinishby}(t, \{p1, p2\})) &= \\
&\quad \text{timerequiredby}(\emptyset), && \text{if } t < 2t_a \\
&\quad \text{timerequiredby}(\{ \langle \text{normal}, p1 \rangle, \langle \text{normal}, p2 \rangle \}), && \text{if } t \geq 2t_a \\
&\quad 0, && \text{if } t < 2t_a \\
&\quad \text{ExecClock}(p1) + \text{ExecClock}(p2) = 2t_a + 1, && \text{if } t \geq 2t_a
\end{aligned}$$

Notice that for $t = 2t_a$...

$$\text{timerequiredby}(\text{mustfinishby}(t, \{p1, p2\})) = 2t_a + 1 > t_a + 1 = (t - t_{\text{event}})$$

This violates the requirement for feasibility, therefore ...

$$\begin{aligned}
\text{feasible}(\{p1, p2\}) &= \text{false} \\
&\rightarrow P_{\text{feasible}}(\{p1, p2\}) = P_{\text{feasible}}(\{p1\}) && (\text{since } P_{\text{leastPV}}(\{p1, p2\}) = \{p2\}) \\
&\quad = \{p1\} && (\text{as shown above}) \\
&\rightarrow P_{\text{scheduled}}(\{p1, p2\}) = \{p1\} && (\text{since } P_{\text{scheduled}}(\{p1, p2\}) = P_{\text{feasible}}(\{p1, p2\}))
\end{aligned}$$

Once again, the set of phases that can feasibly be placed in a schedule based on current knowledge has been determined. Now a single phase must be selected to execute first ...

$$\begin{aligned}
 pmplist &= \text{tobescheduled}(P_{\text{scheduled}}(\{p1, p2\})) \\
 &= \text{tobescheduled}(\{p1\}) \\
 &= \{\langle \text{normal}, p1 \rangle\}
 \end{aligned}$$

$$\begin{aligned}
 DL_{\text{first}}(pmplist) &= DL_{\text{first}}(\{\langle \text{normal}, p1 \rangle\}) \\
 &= 2t_a
 \end{aligned}$$

$$\begin{aligned}
 \text{mustfinishby}(DL_{\text{first}}(pmplist), P_{\text{scheduled}}(\{p1, p2\})) \\
 &= \text{mustfinishby}(2t_a, \{p1\}) \\
 &= \{\langle \text{normal}, p \rangle \mid p \in \text{mustcompleteby}(2t_a, \{p1\})\} \\
 &\quad \text{(assuming } \text{mustcompleteby}(2t_a, \{p1\}) \neq \emptyset) \\
 &= \{\langle \text{normal}, p \rangle \mid p \in \{q \mid [q \in \{p1\} \wedge \text{Deadline}(q) \leq 2t_a]\}\} \\
 &= \{\langle \text{normal}, p \rangle \mid p \in \{p1\}\} \quad \text{(note that } \text{mustcompleteby}(2t_a, \{p1\}) \neq \emptyset) \\
 &= \{\langle \text{normal}, p1 \rangle\}
 \end{aligned}$$

Finally ...

$$\begin{aligned}
 \text{PhaseElect}' &= \text{SelectPhase}(\{p1, p2\}) \\
 &= \text{pickone}(\text{mustfinishby}(DL_{\text{first}}(pmplist), P_{\text{scheduled}}(\{p1, p2\}))) \\
 &= \text{pickone}(\{\langle \text{normal}, p1 \rangle\}) \\
 &= \langle \text{normal}, p1 \rangle
 \end{aligned}$$

Event 6: $t_6 = (t_a - 1)^*$ resume-phase(p1)	S
--	---

event parameters:

$$\begin{aligned}
 t_{\text{event}} &= t_6 = (t_a - 1)^* \\
 p &= p1
 \end{aligned}$$

precondition:

$$\begin{aligned}
 &(\text{RunningPhase} = \text{nullphase}) \wedge (\text{Phase}(\text{PhaseElect}) = p1) \\
 &\quad \wedge (\text{Phase}(\text{PhaseElect}) \neq \text{nullphase}) \wedge (\text{Mode}(\text{PhaseElect}) = \text{normal})
 \end{aligned}$$

≡

$$\begin{aligned}
 &(\text{RunningPhase} = \text{nullphase}) \wedge (\text{Phase}(\langle \text{normal}, p1 \rangle) = p1) \\
 &\quad \wedge (\text{Phase}(\langle \text{normal}, p1 \rangle) \neq \text{nullphase}) \wedge (\text{Mode}(\langle \text{normal}, p1 \rangle) = \text{normal})
 \end{aligned}$$

≡

true

(so the event is accepted)

postconditions:

$$\begin{aligned}
 \text{ResumeTime}'(p1) &= (t_a - 1)^* \\
 \text{RunningPhase}' &= \text{Phase}(\text{PhaseElect}) = \text{Phase}(\langle \text{normal}, p1 \rangle) = p1
 \end{aligned}$$

Event 7: $t_7 = 2t_a$ request-phase(step(0, ∞), 0)	p1
---	----

event parameters:

$$\begin{aligned}
 t_{event} &= t_a = 2t_a \\
 v &= \text{step}(0, \infty) \\
 t_{expected} &= 0 \\
 p &= p1
 \end{aligned}$$

precondition: *true* (so the event is accepted)

postconditions:

$$\begin{aligned}
 Total' &= v3 + \text{Value}(p1)(2t_a) && (\text{since RunningPhase}=p1 \wedge \text{ExecMode}(p1)=\text{normal}) \\
 &= v3 + \text{step}(v1, 2t_a)(2t_a) \\
 &= v3 + v1 \\
 \text{Value}'(p1) &= \text{step}(0, \infty) \\
 \text{ExecClock}'(p1) &= 0 \\
 \text{AbortClock}'(p1) &= 0 \\
 \text{ExecMode}'(p1) &= \text{normal} \\
 \text{PhaseList}' &= \{p1, p2\} - \{p1\} = \{p2\} && (\text{since } t_{expected}=0) \\
 \text{PhaseElect}' &= \text{SelectPhase}(\{p2\}) \\
 \text{RunningPhase}' &= \text{nullphase} && (\text{since } p1 = \text{RunningPhase})
 \end{aligned}$$

Therefore, the history is accepted by the DASA/ND automaton and has a total value of $Total = v1 + v3$.

LBESA Automaton Does Not Accept History H_1 . The first two events are accepted in the same way as they were for DASA/ND. Also, all of the state components, with the possible exception of 'PhaseElect,' are the same for both automata after the first two events. After that, LBESA behaves differently than DASA/ND. The following development shows the behavior of LBESA in detail. (Refer to Section 4.3.2.1 for the definition of the LBESA automaton.)

According to the automaton definition, initially:

$$\begin{aligned}
 Total &= 0 \\
 \text{RunningPhase} &= \text{nullphase} \\
 \text{PhaseElect} &= \langle \text{normal}, \text{nullphase} \rangle \\
 \text{PhaseList} &= \emptyset
 \end{aligned}$$

The following labeled steps demonstrate the acceptance of the first few events in history H_1 and detail the changes in state component values that accompany each event.

Event 1: $t_1 = 0^-$ request-phase(step(v1, $2t_a$), $t_a + 1$) p1
--

event parameters:

$$\begin{aligned}
 t_{event} &= t_1 = 0^- \\
 v &= \text{step}(v1, 2t_a) \\
 t_{expected} &= t_a + 1 \\
 p &= p1
 \end{aligned}$$

precondition: *true* (so the event is accepted)

postconditions:

$Value'(p1) = step(v1, 2t_a)$
 $ExecClock'(p1) = t_a + 1$
 $AbortClock'(p1) = 0$
 $ExecMode'(p1) = normal$
 $PhaseList' = \emptyset \cup \{p1\} = \{p1\}$
 $PhaseElect' = SelectPhase(\{p1\})$

(since $t_{expected} > 0$)

Event 2:	$t_2 = 0^-$	request-phase(step(v2, $2t_a$), t_a)	p2
----------	-------------	--	----

event parameters:

$t_{event} = t_2 = 0^-$
 $v = step(v2, 2t_a)$
 $t_{expected} = t_a$
 $p = p2$

precondition: true

(so the event is accepted)

postconditions:

$Value'(p2) = step(v2, 2t_a)$
 $ExecClock'(p2) = t_a$
 $AbortClock'(p2) = 0$
 $ExecMode'(p2) = normal$
 $PhaseList' = \{p1\} \cup \{p2\} = \{p1, p2\}$
 $PhaseElect' = SelectPhase(\{p1, p2\})$

(since $t_{expected} > 0$)

Event 3:	$t_3 = 0$	request-phase(step(v3, $2t_a - 1$), $t_a - 1$)	p3
----------	-----------	--	----

event parameters:

$t_{event} = t_3 = 0$
 $v = step(v3, 2t_a - 1)$
 $t_{expected} = t_a - 1$
 $p = p3$

precondition: true

(so the event is accepted)

postconditions:

$Value'(p3) = step(v3, 2t_a - 1)$
 $ExecClock'(p3) = t_a - 1$
 $AbortClock'(p3) = 0$
 $ExecMode'(p3) = normal$
 $PhaseList' = \{p1, p2\} \cup \{p3\} = \{p1, p2, p3\}$
 $PhaseElect' = SelectPhase(\{p1, p2, p3\})$

(since $t_{expected} > 0$)

Evaluating $SelectPhase(\{p1, p2, p3\})$...

$SelectPhase(\{p1, p2, p3\}) =$
 $pickone(mustfinishby(DL_{first}(pmplist), P_{scheduled}(\{p1, p2, p3\}))),$
 where
 $pmplist = tobescheduled(P_{scheduled}(\{p1, p2, p3\}))$

$P_{scheduled}(\{p1, p2, p3\}) =$
 $P_{feasible}(P_{scheduled}(\{p1, p3\}) \cup \{p2\})$ (since $P_{lastDL}(\{p1, p2, p3\}) = \{p2\}$)

$P_{scheduled}(\{p1, p3\}) =$
 $P_{feasible}(P_{scheduled}(\{p3\}) \cup \{p1\})$ (since $P_{lastDL}(\{p1, p3\}) = \{p1\}$)

$P_{scheduled}(\{p3\})$
 $= P_{feasible}(P_{scheduled}(\emptyset) \cup \{p3\})$ (since $P_{lastDL}(\{p1\}) = \{p1\}$)
 $= P_{feasible}(\emptyset \cup \{p3\})$
 $= P_{feasible}(\{p3\})$
 $= \{p3\}, \text{ if feasible}(\{p3\})$

$feasible(\{p3\}) = true,$
 iff $(\forall t)[(t \geq t_{event}) \rightarrow timerequiredby(mustfinishby(t, \{p3\})) \leq (t - t_{event})]$

For $t \geq t_{event} \dots$

$mustfinishby(t, \{p3\}) =$
 $\emptyset,$ if $mustcompleteby(t, \{p3\}) = \emptyset$
 $\{ \langle normal, p \rangle \mid p \in mustcompleteby(t, \{p3\}) \},$
 otherwise

$mustcompleteby(t, \{p3\})$
 $= \{ p \mid [p \in \{p3\} \wedge Deadline(p) \leq t] \}$
 $= \emptyset,$ if $t < Deadline(p3) = 2t_a - 1$
 $\{p3\},$ if $t \geq Deadline(p3) = 2t_a - 1$

Therefore ...

$mustfinishby(t, \{p3\})$
 $= \emptyset,$ if $t < 2t_a - 1$
 $\{ \langle normal, p \rangle \mid p \in \{p3\} \},$ otherwise
 $= \emptyset,$ if $t < 2t_a - 1$
 $\{ \langle normal, p3 \rangle \},$ otherwise ($t \geq 2t_a - 1$)

$timerequiredby(mustfinishby(t, \{p3\})) =$
 $= timerequiredby(\emptyset),$ if $t < 2t_a - 1$
 $timerequiredby(\{ \langle normal, p3 \rangle \}),$ if $t \geq 2t_a - 1$
 $= 0,$ if $t < 2t_a - 1$
 $ExecClock(p3) = t_a - 1,$ if $t \geq 2t_a - 1$

Notice that for $t \geq t_{event} = 0$, when $t < 2t_a - 1 \dots$

$timerequiredby(mustfinishby(t, \{p3\})) = 0 \leq (t - t_{event}),$

as required for feasibility

And when $t \geq 2t_a - 1 \dots$

$$timerequiredby(mustfinishby(t, \{p3\})) = t_a - 1 \leq 2t_a - 1 \leq (t - t_{event}),$$

as required for feasibility

Therefore ...

$$feasible(\{p3\}) = true \rightarrow P_{scheduled}(\{p3\}) = \{p3\}$$

Continuing ...

$$\begin{aligned} P_{scheduled}(\{p1, p3\}) &= P_{feasible}(P_{scheduled}(\{p3\}) \cup \{p1\}) && \text{(as shown above)} \\ &= P_{feasible}(\{p3\} \cup \{p1\}) \\ &= P_{feasible}(\{p1, p3\}) \end{aligned}$$

$$\begin{aligned} feasible(\{p1, p3\}) = true, \text{ iff } (\forall t)[(t \geq t_{event}) \\ \rightarrow timerequiredby(mustfinishby(t, \{p1, p3\})) \leq (t - t_{event})] \end{aligned}$$

For $t \geq t_{event}$...

$$\begin{aligned} mustfinishby(t, \{p1, p3\}) = & \begin{cases} 0, & \text{if } mustcompleteby(t, \{p1, p3\}) = 0 \\ \{ \langle normal, p \rangle \mid p \in mustcompleteby(t, \{p1, p3\}) \}, & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} mustcompleteby(t, \{p1, p3\}) &= \{p \mid [p \in \{p1, p3\} \wedge Deadline(p) \leq t]\} \\ &= \begin{cases} 0, & \text{if } t < Deadline(p3) = 2t_a - 1 \\ \{p3\}, & \text{if } Deadline(p3) = 2t_a - 1 \leq t < Deadline(p1) = 2t_a \\ \{p1, p3\}, & \text{if } t \geq Deadline(p1) = 2t_a \end{cases} \end{aligned}$$

Therefore ...

$$\begin{aligned} mustfinishby(t, \{p1, p3\}) &= \begin{cases} 0, & \text{if } t < 2t_a - 1 \\ \{ \langle normal, p \rangle \mid p \in \{p3\} \}, & \text{if } 2t_a - 1 \leq t < 2t_a \\ \{ \langle normal, p \rangle \mid p \in \{p1, p3\} \}, & \text{if } 2t_a \leq t \end{cases} \\ &= \begin{cases} 0, & \text{if } t < 2t_a - 1 \\ \{ \langle normal, p3 \rangle \}, & \text{if } 2t_a - 1 \leq t < 2t_a \\ \{ \langle normal, p1 \rangle, \langle normal, p3 \rangle \}, & \text{if } 2t_a \leq t \end{cases} \end{aligned}$$

$$\begin{aligned} timerequiredby(mustfinishby(t, \{p1, p3\})) &= \begin{cases} timerequiredby(0), & \text{if } t < 2t_a - 1 \\ timerequiredby(\{ \langle normal, p3 \rangle \}), & \text{if } 2t_a - 1 \leq t < 2t_a \\ timerequiredby(\{ \langle normal, p1 \rangle, \langle normal, p3 \rangle \}), & \text{if } 2t_a \leq t \end{cases} \\ &= \begin{cases} 0, & \text{if } t < 2t_a - 1 \\ ExecClock(p3) = t_a - 1, & \text{if } 2t_a - 1 \leq t < 2t_a \\ ExecClock(p1) + ExecClock(p3) = 2t_a, & \text{if } 2t_a \leq t \end{cases} \end{aligned}$$

Notice that for $t \geq t_{event} = 0$, when $t < 2t_a - 1 \dots$

$$timerequiredby(mustfinishby(t, \{p1, p3\})) = 0 \leq (t - t_{event}), \quad \text{as required for feasibility}$$

When $2t_a - 1 \leq t < 2t_a \dots$

$$timerequiredby(mustfinishby(t, \{p1, p3\})) = t_a - 1 \leq 2t_a - 1 \leq (t - t_{event}), \quad \text{as required for feasibility}$$

And when $t \geq 2t_a \dots$

$$timerequiredby(mustfinishby(t, \{p1, p3\})) = 2t_a \leq (t - t_{event}), \quad \text{as required for feasibility}$$

Therefore ...

$$\begin{aligned} feasible(\{p1, p3\}) &= true \\ &\rightarrow P_{feasible}(\{p1, p3\}) = \{p1, p3\} \\ &\rightarrow P_{scheduled}(\{p1, p3\}) = \{p1, p3\} \end{aligned}$$

Continuing ...

$$\begin{aligned} P_{scheduled}(\{p1, p2, p3\}) &= P_{feasible}(P_{scheduled}(\{p1, p3\}) \cup \{p2\}) && \text{(as shown above)} \\ &= P_{feasible}(\{p1, p3\} \cup \{p2\}) \\ &= P_{feasible}(\{p1, p2, p3\}) \end{aligned}$$

To evaluate $P_{feasible}(\{p1, p2, p3\}) \dots$

$$\begin{aligned} feasible(\{p1, p2, p3\}) &= true, \text{ iff } (\forall t) [(t \geq t_{event}) \\ &\rightarrow timerequiredby(mustfinishby(t, \{p1, p2, p3\})) \leq (t - t_{event})] \end{aligned}$$

For $t \geq t_{event} \dots$

$$\begin{aligned} mustfinishby(t, \{p1, p2, p3\}) &= \\ &\begin{cases} 0, & \text{if } mustcompleteby(t, \{p1, p2, p3\}) = 0 \\ \{ \langle normal, p \rangle \mid p \in mustcompleteby(t, \{p1, p2, p3\}) \}, & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} mustcompleteby(t, \{p1, p2, p3\}) &= \{p \mid [p \in \{p1, p2, p3\} \wedge Deadline(p) \leq t]\} \\ &= \begin{cases} 0, & \text{if } t < Deadline(p3) = 2t_a - 1 \\ \{p3\}, & \text{if } Deadline(p3) = 2t_a - 1 \leq t < Deadline(p1) = 2t_a \\ \{p1, p2, p3\}, & \text{if } t \geq Deadline(p1) = 2t_a \end{cases} \end{aligned}$$

Therefore ...

$$\begin{aligned}
& \text{mustfinishby}(t, \{p1, p2, p3\}) \\
&= 0, & \text{if } t < 2t_a - 1 \\
& \{ \langle \text{normal}, p \rangle \mid p \in \{p3\} \}, & \text{if } 2t_a - 1 \leq t < 2t_a \\
& \{ \langle \text{normal}, p \rangle \mid p \in \{p1, p2, p3\} \}, & \text{if } 2t_a \leq t \\
&= 0, & \text{if } t < 2t_a - 1 \\
& \{ \langle \text{normal}, p3 \rangle \}, & \text{if } 2t_a - 1 \leq t < 2t_a \\
& \{ \langle \text{normal}, p1 \rangle, \langle \text{normal}, p2 \rangle, \langle \text{normal}, p3 \rangle \}, & \text{if } 2t_a \leq t \\
& \text{timerequiredby}(\text{mustfinishby}(t, \{p1, p2, p3\})) = \\
&= \text{timerequiredby}(0), & \text{if } t < 2t_a - 1 \\
& \text{timerequiredby}(\{ \langle \text{normal}, p3 \rangle \}), & \text{if } 2t_a - 1 \leq t < 2t_a \\
& \text{timerequiredby}(\{ \langle \text{normal}, p1 \rangle, \langle \text{normal}, p2 \rangle, \langle \text{normal}, p3 \rangle \}), & \text{if } 2t_a \leq t \\
&= 0, & \text{if } t < 2t_a - 1 \\
& \text{ExecClock}(p3) = t_a - 1, & \text{if } 2t_a - 1 \leq t < 2t_a \\
& \text{ExecClock}(p1) + \text{ExecClock}(p2) + \text{ExecClock}(p3) = 3t_a, & \text{if } 2t_a \leq t
\end{aligned}$$

Notice that for $t = 2t_a \dots$

$$\text{timerequiredby}(\text{mustfinishby}(t, \{p1, p2, p3\})) = 3t_a > 2t_a = (t - t_{\text{event}})$$

This violates the requirement for feasibility, therefore ...

$$\begin{aligned}
& \text{feasible}(\{p1, p2, p3\}) = \text{false} \\
& \rightarrow P_{\text{feasible}}(\{p1, p2, p3\}) = P_{\text{feasible}}(\{p1, p2\}) \quad (\text{since } P_{\text{feasible}}(\{p1, p2, p3\}) = \{p3\})
\end{aligned}$$

To evaluate $P_{\text{feasible}}(\{p1, p2\}) \dots$

$$\begin{aligned}
& \text{feasible}(\{p1, p2\}) = \text{true, iff } (\forall t) \{ (t \geq t_{\text{event}}) \\
& \rightarrow \text{timerequiredby}(\text{mustfinishby}(t, \{p1, p2\})) \leq (t - t_{\text{event}}) \}
\end{aligned}$$

For $t \geq t_{\text{event}} \dots$

$$\begin{aligned}
& \text{mustfinishby}(t, \{p1, p2\}) = \\
& 0, & \text{if } \text{mustcompleteby}(t, \{p1, p2\}) = 0 \\
& \{ \langle \text{normal}, p \rangle \mid p \in \text{mustcompleteby}(t, \{p1, p2\}) \}, & \text{otherwise} \\
& \text{mustcompleteby}(t, \{p1, p2\}) \\
&= \{ p \mid [p \in \{p1, p2\} \wedge \text{Deadline}(p) \leq t] \} \\
&= 0, & \text{if } t < \text{Deadline}(p1) = \text{Deadline}(p2) = 2t_a \\
& \{p1, p2\}, & \text{if } t \geq \text{Deadline}(p1) = \text{Deadline}(p2) = 2t_a
\end{aligned}$$

Therefore ...

$$\begin{aligned}
& \text{mustfinishby}(t, \{p1, p2\}) \\
&= 0, & \text{if } t < 2t_a \\
& \{ \langle \text{normal}, p \rangle \mid p \in \{p1, p2\} \}, & \text{otherwise} \\
&= 0, & \text{if } t < 2t_a \\
& \{ \langle \text{normal}, p1 \rangle, \langle \text{normal}, p2 \rangle \}, & \text{otherwise } (t \geq 2t_a) \\
& \text{timerequiredby}(\text{mustfinishby}(t, \{p1, p2\})) = \\
&= \text{timerequiredby}(0), & \text{if } t < 2t_a \\
& \text{timerequiredby}(\{ \langle \text{normal}, p1 \rangle, \langle \text{normal}, p2 \rangle \}), & \text{if } t \geq 2t_a \\
&= 0, & \text{if } t < 2t_a \\
& \text{ExecClock}(p1) + \text{ExecClock}(p2) = 2t_a + 1, & \text{if } t \geq 2t_a
\end{aligned}$$

Notice that for $t = 2t_a \dots$

$$\text{timerequiredby}(\text{mustfinishby}(t, \{p1, p2\})) = 2t_a + 1 > 2t_a = (t - t_{\text{event}})$$

This violates the requirement for feasibility, therefore $\dots$

$$\begin{aligned}
& \text{feasible}(\{p1, p2\}) = \text{false} \\
& \rightarrow P_{\text{feasible}}(\{p1, p2\}) = P_{\text{feasible}}(\{p1\}) & (\text{since } P_{\text{leastPV}}(\{p1, p2\}) = \{p2\})
\end{aligned}$$

To evaluate $P_{\text{feasible}}(\{p1\}) \dots$

$$\begin{aligned}
& \text{feasible}(\{p1\}) = \text{true}, \\
& \text{iff } (\forall t)[(t \geq t_{\text{event}}) \rightarrow \text{timerequiredby}(\text{mustfinishby}(t, \{p1\})) \leq (t - t_{\text{event}})]
\end{aligned}$$

For $t \geq t_{\text{event}} \dots$

$$\begin{aligned}
& \text{mustfinishby}(t, \{p1\}) = \\
& 0, & \text{if } \text{mustcompleteby}(t, \{p1\}) = 0 \\
& \{ \langle \text{normal}, p \rangle \mid p \in \text{mustcompleteby}(t, \{p1\}) \}, & \text{otherwise} \\
& \text{mustcompleteby}(t, \{p1\}) \\
&= \{ p \mid [p \in \{p1\} \wedge \text{Deadline}(p) \leq t] \} \\
&= 0, & \text{if } t < \text{Deadline}(p1) = 2t_a \\
& \{p1\}, & \text{if } t \geq \text{Deadline}(p1) = 2t_a
\end{aligned}$$

Therefore $\dots$

$$\begin{aligned}
& \text{mustfinishby}(t, \{p1\}) \\
& \quad = 0, & \text{if } t < 2t_a \\
& \quad \{ \langle \text{normal}, p \rangle \mid p \in \{p1\} \}, & \text{otherwise} \\
& \quad = 0, & \text{if } t < 2t_a \\
& \quad \{ \langle \text{normal}, p1 \rangle \}, & \text{otherwise } (t \geq 2t_a) \\
& \text{timerequiredby}(\text{mustfinishby}(t, \{p1\})) = \\
& \quad = \text{timerequiredby}(0), & \text{if } t < 2t_a \\
& \quad \text{timerequiredby}(\{ \langle \text{normal}, p1 \rangle \}), & \text{if } t \geq 2t_a \\
& \quad = 0, & \text{if } t < 2t_a \\
& \quad \text{ExecClock}(p1) = t_a + 1, & \text{if } t \geq 2t_a
\end{aligned}$$

Notice that for $t \geq t_{\text{event}} = 0$, when $t < 2t_a \dots$

$$\text{timerequiredby}(\text{mustfinishby}(t, \{p1\})) = 0 \leq (t - t_{\text{event}}), \quad \text{as required for feasibility}$$

And when $t \geq 2t_a \dots$

$$\text{timerequiredby}(\text{mustfinishby}(t, \{p1\})) = t_a + 1 \leq 2t_a \leq (t - t_{\text{event}}), \quad \text{as required for feasibility}$$

Therefore ...

$$\text{feasible}(\{p1\}) = \text{true} \rightarrow P_{\text{feasible}}(\{p1\}) = \{p1\}$$

Putting this together ...

$$\begin{aligned}
P_{\text{scheduled}}(\{p1, p2, p3\}) &= P_{\text{feasible}}(\{p1, p2, p3\}) \\
&= P_{\text{feasible}}(\{p1, p2\}) & (\text{since } \text{feasible}(\{p1, p2, p3\}) = \text{false} \wedge P_{\text{least}P1}(\{p1, p2, p3\}) = \{p3\}) \\
&= P_{\text{feasible}}(\{p1\}) & (\text{since } \text{feasible}(\{p1, p2\}) = \text{false} \wedge P_{\text{least}P1}(\{p1, p2\}) = \{p2\}; \\
&= \{p1\} & (\text{as shown above})
\end{aligned}$$

At this point, the set of phases that can be feasibly executed has been determined. Now to decide which phase to be executed first ...

$$\begin{aligned}
\text{pmplist} &= \text{tocheduled}(P_{\text{scheduled}}(\{p1, p2, p3\})) \\
&= \text{tocheduled}(\{p1\}) \\
&= \{ \langle \text{normal}, p1 \rangle \}
\end{aligned}$$

$$\begin{aligned}
DL_{\text{first}}(\text{pmplist}) &= DL_{\text{first}}(\{ \langle \text{normal}, p1 \rangle \}) \\
&= 2t_a
\end{aligned}$$

$$\begin{aligned}
& \text{mustfinishby}(DL_{\text{first}}(\text{pmplist}), P_{\text{scheduled}}(\{p1, p2, p3\})) \\
&= \text{mustfinishby}(2t_a, \{p1\}) \\
&= \{ \langle \text{normal}, p \rangle \mid p \in \text{mustcompleteby}(2t_a, \{p1\}) \} & (\text{assuming } \text{mustcompleteby}(2t_a, \{p1\}) \neq \emptyset) \\
&= \{ \langle \text{normal}, p \rangle \mid p \in \{q \mid [q \in \{p1\} \wedge \text{Deadline}(q) \leq 2t_a] \} \} \\
&= \{ \langle \text{normal}, p \rangle \mid p \in \{p1\} \} & (\text{note that: } \text{mustcompleteby}(2t_a, \{p1\}) \neq \emptyset) \\
&= \{ \langle \text{normal}, p1 \rangle \}
\end{aligned}$$

Finally ...

$$\begin{aligned} \text{PhaseElect}' &= \text{SelectPhase}(\{p1, p2, p3\}) \\ &= \text{pickone}(\text{mustfinishby}(DL_{first}(pmlist)P_{scheduled}(\{p1, p2, p3\}))) \\ &= \text{pickone}(\{\langle \text{normal}, p1 \rangle\}) \\ &= \langle \text{normal}, p1 \rangle \end{aligned}$$

Event 4: $t_4 = 0^+$ resume-phase(p3) S
--

event parameters:

$$\begin{aligned} t_{event} &= t_4 = 0^+ \\ p &= p3 \end{aligned}$$

precondition:

$$\begin{aligned} &(\text{RunningPhase} = \text{nullphase}) \wedge (\text{Phase}(\text{PhaseElect}) = p3) \\ &\wedge (\text{Phase}(\text{PhaseElect}) \neq \text{nullphase}) \wedge (\text{Mode}(\text{PhaseElect}) = \text{normal}) \end{aligned}$$

$$\equiv$$

$$\begin{aligned} &(\text{RunningPhase} = \text{nullphase}) \wedge (\text{Phase}(\langle \text{normal}, p1 \rangle) = p3) \\ &\wedge (\text{Phase}(\langle \text{normal}, p1 \rangle) \neq \text{nullphase}) \wedge (\text{Mode}(\langle \text{normal}, p1 \rangle) = \text{normal}) \end{aligned}$$

$$\equiv$$

false,

(since $\text{Phase}(\langle \text{normal}, p1 \rangle) = p1 \neq p3$)

Since the precondition is not satisfied, the event cannot be accepted.

Therefore, history H_1 is not accepted by the LBESA automaton.

LBESA cannot accept any history that begins with Events (1)-(3) and that has only those three phases, with the already specified time-value functions and computation time requirements, that will yield a total value greater than $v1$.

This proof will be carried out by identifying all of the histories that LBESA can accept under these circumstances. The total value resulting from each of these histories will then be examined to demonstrate that none is greater than $v1$.

To begin to identify the histories that are accepted by LBESA, notice that, given Events (1)-(3), LBESA will behave exactly as described in the preceding section. After accepting Event (3), the third event in this sequence, the only events whose preconditions are satisfied are:

1. any 'request-phase'
2. 'resume-phase(p1)'

Examine the first possibility — any 'request-phase' event — more closely. Let p_x denote the phase originating a 'request phase' event. If $p_x \notin \{p1, p2, p3\}$, then p_x is a new phase. But this violates the

assertion that the only histories being considered consist solely of events associated with phases p_1 , p_2 , and p_3 . Therefore, p_x must be a member of $\{p_1, p_2, p_3\}$.

Also, notice that after accepting Events (1)-(3), $RunningPhase = nullphase$, which is not a member of $\{p_1, p_2, p_3\}$. Consequently, the postconditions of a 'request-phase'(v_x, t_x) p_x event are:

$$\begin{aligned} Value'(p_x) &= v_x \\ ExecClock'(p_x) &= t_x \\ AbortClock'(p_x) &= 0 \\ ExecMode'(p_x) &= normal \\ PhaseList' &= PhaseList \cup \{p_x\} \text{ or } PhaseList - \{p_x\} \\ PhaseElect' &= SelectPhase(PhaseList') \end{aligned}$$

Notice that these postconditions serve only to alter or reiterate the scheduling parameters of the already defined phases (possibly removing one of the phases from consideration from scheduling at the same time and potentially selecting a new $PhaseElect$ to reflect these changes). If the scheduling parameters are altered, this violates the assertion that the automaton will consider only the time-value functions and expected computation times already specified for the three phases by the first three events. Consequently, the only 'request-phase' events that LBESA can accept at this point reiterate the scheduling parameters for $p_x \in \{p_1, p_2, p_3\}$. (Hereafter, 'request-phase' events that serve to reiterate previously defined scheduling parameters may be referred to as *reiterative 'request-phase' events*.) Furthermore, notice that although such 'request-phase' events do not alter the scheduling parameters for a phase — they merely reiterate them — there is a potential effect of these events on the automaton state component $PhaseElect$, which is set equal to $SelectPhase(PhaseList')$ as a postcondition of each 'request-phase' event. The function $SelectPhase()$ is dependent on t_{event} , which increases during the course of any history.

To examine the effect of a 'request-phase' on $PhaseElect$ consider first the effect on the value of $P_{scheduled}(\{p_1, p_2, p_3\})$ as a function of t_{event} . Of course, $t_{event} > 0$ since only legal histories are under consideration here, and the third event occurred at time $t_2 = 0$. With that in mind, expand the value of $P_{scheduled}(\{p_1, p_2, p_3\})$ as follows:

$$\begin{aligned} P_{scheduled}(\{p_1, p_2, p_3\}) &= P_{feasible}(P_{scheduled}(\{p_1, p_3\}) \cup \{p_2\}) && (\text{since } P_{lastDL}(\{p_1, p_2, p_3\}) = \{p_2\}) \\ &= P_{feasible}(P_{feasible}(P_{scheduled}(\{p_3\}) \cup \{p_1\}) \cup \{p_2\}) && (\text{since } P_{lastDL}(\{p_1, p_3\}) = \{p_1\}) \\ &= P_{feasible}(P_{feasible}(P_{feasible}(P_{scheduled}(\emptyset) \cup \{p_3\}) \cup \{p_1\}) \cup \{p_2\}) && (\text{since } P_{lastDL}(\{p_3\}) = \{p_3\}) \\ &= P_{feasible}(P_{feasible}(P_{feasible}(\emptyset \cup \{p_3\}) \cup \{p_1\}) \cup \{p_2\}) \\ &= P_{feasible}(P_{feasible}(P_{feasible}(\{p_3\}) \cup \{p_1\}) \cup \{p_2\}) \\ P_{feasible}(\{p_3\}) &= && \\ &\quad \{p_3\}, && \text{if } feasible(\{p_3\}) \\ &\quad \emptyset, && \text{otherwise} \end{aligned}$$

Several feasibility conditions like this will have to be evaluated in the following section of the proof. Therefore, a general result will be derived here that can be applied to any of the simple cases that follow. Consider a phase p with automaton state components:

$Value(p) = step(v, t_{DL})$
 $ExecClock(p) = t_{required}$
 $AbortClock(p) = 0$
 $ExecMode(p) = normal$

Notice that $p1$, $p2$, and $p3$ all satisfy this profile at this point in the automaton's examination of any history that it accepts. Then ...

$$feasible(\{p\}) = true, \text{ iff } (\forall t)[(t \geq t_{event}) \rightarrow timerequiredby(mustfinishby(t, \{p\})) \leq (t - t_{event})]$$

For $t_{event} > t \dots$

$$\begin{aligned}
 timerequiredby(mustfinishby(t, \{p\})) \\
 &= timerequiredby(0), && \text{if } mustcompleteby(t, \{p\}) = 0 \\
 &timerequiredby(\{ \langle normal, q \rangle \mid q \in mustcompleteby(t, \{p\}) \}), && \text{otherwise}
 \end{aligned}$$

$$mustcompleteby(t, \{p\}) = \{q \mid [q \in \{p\} \wedge Deadline(q) \leq t]\}$$

$$\begin{aligned}
 &= 0, && \text{if } t < t_{DL} \\
 &\{p\}, && \text{if } t \geq t_{DL}
 \end{aligned}$$

Therefore,

$$\begin{aligned}
 timerequiredby(mustfinishby(t, \{p\})) \\
 &= timerequiredby(0), && \text{if } t < t_{DL} \\
 &timerequiredby(\{ \langle normal, p \rangle \}), && \text{if } t \geq t_{DL} \\
 &= 0, && \text{if } t < t_{DL} \\
 &t_{required}, && \text{if } t \geq t_{DL}
 \end{aligned}$$

If $feasible(\{p\}) = true$, then, by definition, for any $t \geq t_{event} \dots$

$$timerequiredby(mustfinishby(t, \{p\})) \leq (t - t_{event})$$

For the cases where $t < t_{DL}$, this relation is trivially satisfied since the left-hand side of the relation is equal to zero and the right-hand side is greater than or equal to zero by definition ($(t \geq t_{event} \rightarrow (t - t_{event}) \geq 0)$). For the cases where $t \geq t_{DL} \dots$

$$\begin{aligned}
 t_{required} &\leq t - t_{event} \\
 \rightarrow t_{event} &\leq t - t_{required} \leq t_{DL} - t_{required} && (\text{since } t \geq t_{DL})
 \end{aligned}$$

Applying this general result to each of the three phases under consideration yields:

- $feasible(\{p1\}) = true$, iff $t_{event} \leq 2t_a - (t_a + 1) = t_a - 1$
- $feasible(\{p2\}) = true$, iff $t_{event} \leq 2t_a - t_a = t_a$
- $feasible(\{p3\}) = true$, iff $t_{event} \leq (2t_a - 1) - (t_a - 1) = t_a$

Using this information in the previously derived expression for $P_{feasible}(\{p3\})$ yields ...

$$\begin{aligned}
P_{feasible}(\{p3\}) &= \{p3\}, && \text{if } feasible(\{p3\}) \\
&0, && \text{otherwise} \\
\\
&= \{p3\}, && \text{if } 0 \leq t_{event} \leq t_a \\
&0, && \text{if } t_a < t_{event}
\end{aligned}$$

$$\begin{aligned}
P_{scheduled}(\{p1, p2, p3\}) &= P_{feasible}(P_{feasible}(\{p3\} \cup \{p1\}) \cup \{p2\}), && \text{if } 0 \leq t_{event} \leq t_a \\
&P_{feasible}(P_{feasible}(\emptyset \cup \{p1\}) \cup \{p2\}), && \text{if } t_a < t_{event} \\
\\
&= P_{feasible}(P_{feasible}(\{p1, p3\}) \cup \{p2\}), && \text{if } 0 \leq t_{event} \leq t_a \\
&P_{feasible}(P_{feasible}(\{p1\}) \cup \{p2\}), && \text{if } t_a < t_{event}
\end{aligned}$$

For $t_a < t_{event} \dots$

$$\begin{aligned}
P_{scheduled}(\{p1, p2, p3\}) &= P_{feasible}(P_{feasible}(\{p1\}) \cup \{p2\}) \\
&= P_{feasible}(\emptyset \cup \{p2\}) && (\text{since } feasible(\{p1\}) = \text{false for } t_a < t_{event}) \\
&= P_{feasible}(\{p2\}) \\
&= \emptyset && (\text{since } feasible(\{p2\}) = \text{false for } t_a < t_{event})
\end{aligned}$$

Consider the other case in the derivation of $P_{scheduled}(\{p1, p2, p3\})$, where $0 \leq t_{event} \leq t_a \dots$

$$\begin{aligned}
P_{\text{scheduled}}(\{p1, p2, p3\}) &= P_{\text{feasible}}(P_{\text{feasible}}(\{p1, p3\}) \cup \{p2\}) \\
&= P_{\text{feasible}}(\{p1, p3\} \cup \{p2\}), & \text{if } t_{\text{event}}=0 & \quad (\text{since } P_{\text{feasible}}(\{p1, p3\}) = \{p1, p3\}) \\
&\quad P_{\text{feasible}}(P_{\text{feasible}}(\{p1\}) \cup \{p2\}), & \text{if } 0 < t_{\text{event}} \leq t_a & \quad (\text{since } P_{\text{feasible}}(\{p1, p3\}) = P_{\text{feasible}}(\{p1\})) \\
&= P_{\text{feasible}}(\{p1, p2, p3\}), & \text{if } t_{\text{event}}=0 & \\
&\quad P_{\text{feasible}}(\{p1\} \cup \{p2\}), & \text{if } 0 < t_{\text{event}} \leq t_a - 1 & \quad (\text{since } \text{feasible}(\{p1\}) = \text{true iff } t_{\text{event}} \leq t_a - 1) \\
&\quad P_{\text{feasible}}(0 \cup \{p2\}), & \text{if } t_a - 1 < t_{\text{event}} \leq t_a & \\
&= P_{\text{feasible}}(\{p1, p2, p3\}), & \text{if } t_{\text{event}}=0 & \\
&\quad P_{\text{feasible}}(\{p1, p2\}), & \text{if } 0 < t_{\text{event}} \leq t_a - 1 & \\
&\quad P_{\text{feasible}}(\{p2\}), & \text{if } t_a - 1 < t_{\text{event}} \leq t_a & \\
&= P_{\text{feasible}}(\{p1, p2, p3\}), & \text{if } t_{\text{event}}=0 & \\
&\quad P_{\text{feasible}}(\{p1\}), & \text{if } 0 < t_{\text{event}} \leq t_a - 1 & \quad (\text{since } P_{\text{feasible}}(\{p1, p2\}) = P_{\text{feasible}}(\{p1\})) \\
&\quad P_{\text{feasible}}(\{p2\}), & \text{if } t_a - 1 < t_{\text{event}} \leq t_a & \\
&= \{p1\}, & \text{if } t_{\text{event}}=0 & \quad (\text{as shown previously}) \\
&\quad \{p1\}, & \text{if } 0 < t_{\text{event}} \leq t_a - 1 & \quad (\text{since } \text{feasible}(\{p1\}) = \text{true iff } t_{\text{event}} \leq t_a - 1) \\
&\quad \{p2\}, & \text{if } t_a - 1 < t_{\text{event}} \leq t_a & \quad (\text{since } \text{feasible}(\{p2\}) = \text{true iff } t_{\text{event}} \leq t_a)
\end{aligned}$$

Putting it all together . . .

$$\begin{aligned}
P_{\text{scheduled}}(\{p1, p2, p3\}) &= \\
&\quad \{p1\}, & \text{if } 0 \leq t_{\text{event}} \leq t_a - 1 & \\
&\quad \{p2\}, & \text{if } t_a - 1 < t_{\text{event}} \leq t_a & \\
&\quad 0, & \text{if } t_a < t_{\text{event}} &
\end{aligned}$$

Remember that, by definition:

$$\begin{aligned}
\text{SelectPhase}(\{p1, p2, p3\}) &= \\
&\quad \text{pickone}(\text{mustfinishby}(DL_{\text{first}}(\text{pmplist}), P_{\text{scheduled}}(\{p1, p2, p3\}))), \\
&\quad \text{where} \\
&\quad \text{pmplist} = \text{tobescheduled}(P_{\text{scheduled}}(\{p1, p2, p3\}))
\end{aligned}$$

As a consequence of these last two points:

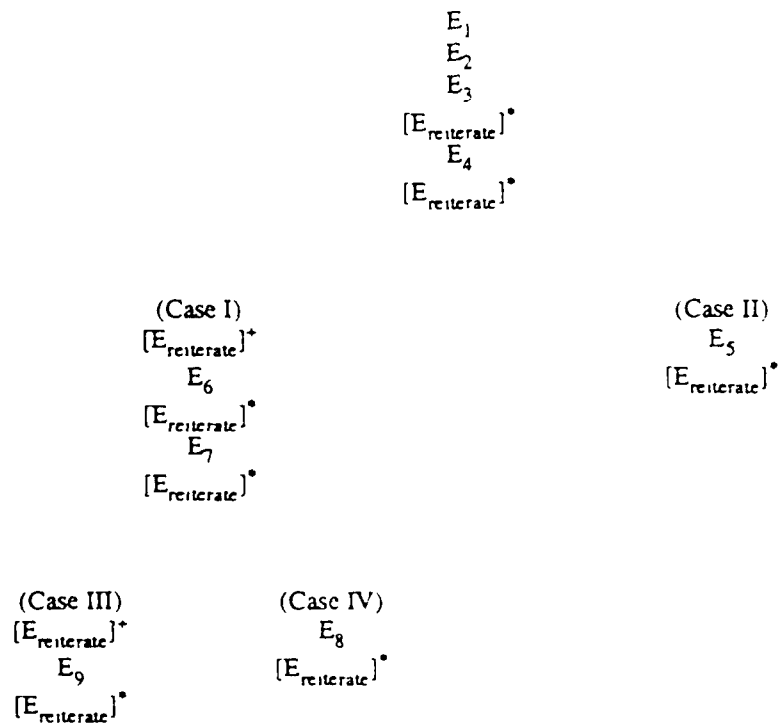
$$\begin{aligned}
\text{SelectPhase}(\{p1, p2, p3\}) &= \\
&\quad \langle \text{normal}, p1 \rangle, & \text{if } 0 \leq t_{\text{event}} \leq t_a - 1 & \\
&\quad \langle \text{normal}, p2 \rangle, & \text{if } t_a - 1 < t_{\text{event}} \leq t_a & \\
&\quad \langle \text{normal}, \text{nullphase} \rangle, & \text{if } t_a < t_{\text{event}} &
\end{aligned}$$

The outcome of this portion of the analysis is that any number of 'request-phase' events can occur to

reiterate scheduling parameters of the three phases of concern. These events will be accepted by the LBESA automaton and the *PhaseElect* state component will have its value changed as indicated above for $PhaseElect = SelectPhase(\{p1, p2, p3\})$. The (possibly empty) sequence of scheduling parameter reiterations may be broken by a 'resume-phase' event for phase *PhaseElect* at any time.

For reasons similar to those offered earlier, this 'resume-phase' event may be followed by any number of other 'request-phase' events for the two phases that are not executing. Once again, these 'request-phase' events may change the value of the *PhaseElect* phase component. In fact, the first 'request-phase' event occurring after the 'resume-phase' may cause a change in value in *PhaseElect*, thus potentially triggering a preemption.

Since it is difficult to follow a narrative description of all of the potential histories that may be accepted by LBESA, the following approach is taken. Consider the diagram below, where each labeled item is an event, "E*" indicates one or more occurrences of the expression "E", and "E*" indicates zero or more occurrences of the expression "E". (The labels "(Case X)" are merely used in the ensuing discussion to refer to specific branches of the diagram.)



where

$E_1:$	$t_1 = 0^-$	request-phase(step(v1, $2t_a$), t_a+1)	p1
$E_2:$	$t_2 = 0^-$	request-phase(step(v2, $2t_a$), t_a)	p2
$E_3:$	$t_3 = 0$	request-phase(step(v3, $2t_a-1$), t_a-1)	p3
$E_4:$	t_4	resume-phase(p_{first})	S
$E_5:$	t_5	request-phase(step(0, ∞), 0)	p_{first}
$E_6:$	t_6	preempt-phase(p_{first})	S
$E_7:$	t_7	resume-phase(p_{second})	S
$E_8:$	t_8	request-phase(step(0, ∞), 0)	p_{second}
$E_9:$	t_9	preempt-phase(p_{second})	S
$E_{reiterate}:$	'request-phase' reiterating scheduling parameters for a phase other than <i>RunningPhase</i>		

To interpret the above diagram, each history accepted by LBESA begins with the first event, E_1 , which is on the first line of the diagram. To trace an individual history accepted by LBESA, begin with the top line and proceed down one line at a time. Where there are branches, choose one path or the other and continue to move down through the diagram. The history may be terminated at any time³⁰

To demonstrate that the diagram is correct, that is, that it incorporates all of the legal histories that LBESA will accept, consider the following rationale.

As was discussed earlier, a 'request-phase' event may be accepted at any time, as long as it serves only to reiterate the already established scheduling parameters for a phase. As a result, the diagram indicates that such events, labeled $[E_{reiterate}]^*$, may occur between any other two events in a history.

As was also shown earlier, an examination of the preconditions of the various potential events indicates that the only event that may be accepted after E_1 , E_2 , E_3 , and any other reiterative 'request-phase' events is a 'resume-phase' event to start the execution of the phase that is currently designated *PhaseElect* (as long as *PhaseElect* is not the *nullphase*). Hence, E_4 can only be a 'resume-phase' event.

There are two possible courses that may be followed after E_4 : (a) the phase may be preempted (Case I) or (b) it may complete execution (Case II). In the latter case, if the phase runs to completion, then it will originate a 'request-phase' event to signal that circumstance. This event will always be accepted because its precondition is simply *true*. In Case I, examination of the precondition for a 'preempt-phase' event indicates that a preemption can only occur if *RunningPhase* is not the same as *PhaseElect* and is not the *nullphase*. The postconditions of event E_4 guarantee that *RunningPhase* is not the *nullphase*. Hence, if a 'request-phase' event following E_4 yielded a value of *PhaseElect* different from *RunningPhase*, then, according to the previous analysis of *SelectPhase*($\{p1, p2, p3\}$), the only possibilities are:

1. E_4 resumed p1, and *PhaseElect* subsequently becomes either p2 or *nullphase*, or
2. E_4 resumed p2, and *PhaseElect* subsequently becomes the *nullphase*

Currently, the assumption is made that the required computation time for a phase is known exactly. Whenever the phase designated by *PhaseElect* is resumed immediately after a 'request-phase' event, it will

³⁰At any time after the third event, that is. By definition, the only histories being considered are those that begin with events E_1 , E_2 , and E_3 , in that order.

be able to meet its deadline if it runs uninterrupted because a test of feasibility was carried out that verified exactly that fact. However, if time is allowed to elapse between the 'request-phase' and the 'resume-phase' events, it is possible that it is no longer feasible to execute *PhaseElect* by the time it is actually initiated³¹. Subsequent 'request-phase' events serve to indicate that fact by selecting a *PhaseElect* other than *RunningPhase*, thereby setting the stage for a preemption.

Consider the next non-'request-phase' event to be accepted by LBESA under Case II in the diagram. If the first phase to execute, p_{first} , completes execution, it signals this fact by originating event E_5 . Then the subsequent evaluation of either *PhaseElect*=*SelectPhase*({ p_2, p_3 }) (in the case where p_{first} was p_1) or *PhaseElect*=*SelectPhase*({ p_1, p_3 }) (in the case where p_{first} was p_2) yields *PhaseElect* = <normal, nullphase>. Therefore, no subsequent 'resume-phase' event can be accepted by the automaton since the necessary precondition cannot be satisfied. Also, since *RunningPhase* = nullphase following E_5 , no new 'preempt-phase' event can be accepted either. So, except for reiterative 'request-phase' events, no further events can be accepted in these particular histories.

In Case I, where the first phase to execute was preempted, this fact was indicated by event E_6 . As one of its postconditions, E_6 set *RunningPhase* = nullphase. The next event in any history accepted by LBESA, other than reiterative 'request-phase' events, cannot be another 'preempt-phase' event because that would require *RunningPhase* $\neq$ nullphase. Therefore, if any event other than a reiterative 'request-phase' event is to be accepted by LBESA, it must be a 'resume-phase'. In order to have such an event occur, *PhaseElect* must, as a precondition, be non-nullphase. This can result from a reiterative 'request-phase' event according to an analysis similar to the one done above.

Finally, if E_7 , a 'resume-phase' event, is accepted in a history, then the situation and analysis is almost identical to the one that was examined after event E_4 , the previous 'resume-phase'. Once again, the resumed phase, p_{second} in this case, can either be preempted (Case III) or run to completion (Case IV), and the circumstances for each of these outcomes is exactly analogous to those given earlier for E_4 . However, the earlier examination of *SelectPhase*({ p_1, p_2, p_3 }) shows that there is no possible successor phase to execute following either E_8 or E_9 . In both cases, this is due to the fact that p_{second} must be p_2 and *PhaseElect* = *SelectPhase*({ p_1, p_2, p_3 }) and *PhaseElect* = *SelectPhase*({ p_1, p_3 }) both yield *PhaseElect* = <normal, nullphase>, which will not permit a subsequent 'resume-phase' event to be accepted by LBESA.

While the above arguments demonstrate that the earlier diagram incorporates all of the legal histories that may be accepted by LBESA, they do not reveal all of the factors involved in making the histories acceptable. In particular, there are constraints on the times at which certain events occur, above and beyond those that apply to any legal history, that must be satisfied to obtain certain histories. For instance, depending on the

³¹Intuitively, this can be thought of as reflecting a latency issue. In effect, the scheduler determines what can be feasibly completed in the available time from the instant at which a scheduling decision is made. However, if the latency encountered in actually dispatching the next phase is large enough, then, by the time it has dispatched the phase, the set of phases that is feasible has changed. Notice that it is possible to specify this latency and apply certain restrictions to histories in order to model and accommodate the latency. Also, it is possible to alter the algorithm embedded in the automaton to handle this latency when it is determining *PhaseElect*.

timing of events, there is the possibility of executing zero, one, or two phases during the course of a history. The following list specifies the time constraints that must be satisfied by various events to obtain given histories:

1. if the history includes event E_4 , then p_{first} may be either p_1 or p_2 ; if it is to be p_1 , then t_{event} for the 'request-phase' immediately preceding E_4 must satisfy:

$$0 \leq t_{\text{event}} \leq t_a - 1$$

- if p_{first} is to be p_2 , then t_{event} for the 'request-phase' immediately preceding E_4 must satisfy:

$$t_a - 1 < t_{\text{event}} \leq t_a$$

2. if the history includes event E_6 (Case I), then either:

- a. $p_{\text{first}} = p_1$ — in this case, t_4 , the time at event which E_4 occurred, must have satisfied:

$$t_a - 1 < t_4$$

- b. $p_{\text{first}} = p_2$ — in this case, t_4 , the time at event which E_4 occurred, must have satisfied:

$$t_a < t_4$$

3. if the history includes event E_5 (Case II), then either³²:

- a. $p_{\text{first}} = p_1$ — in this case, t_5 , the time at event which E_5 occurs, must satisfy:

$$t_5 = t_4 + (t_a + 1)$$

since required computation time is known accurately.

- b. $p_{\text{first}} = p_2$ — in this case, t_5 , the time at event which E_5 occurs, must satisfy:

$$t_5 = t_4 + t_a$$

since required computation time is known accurately.

4. if the history includes event E_7 , then p_{first} must be p_1 and p_{second} must be p_2 . In addition, t_{event} for the 'request-phase' immediately preceding E_7 must satisfy:

$$t_a - 1 < t_{\text{event}} \leq t_a$$

5. if the history includes event E_8 (Case IV), then t_8 , the time at event which E_8 occurs, must satisfy³³:

$$t_8 = t_7 + t_a$$

since required computation time is known accurately.

6. if the history includes event E_9 (Case III), then, since $p_{\text{first}} = p_2$, t_7 , the time at event which E_7 occurred, must have satisfied:

$$t_a < t_7$$

Once all of the histories that are accepted by LBESA have been enumerated, their respective values can also be enumerated. To that end, the table shown in Figure 4-7 puts all of the preceding pieces of the argument together. It lists all of the histories accepted by LBESA that start with events E_1 , E_2 , and E_3 , along with their corresponding values.

³²This is actually a requirement of any legal history. It is explicitly listed here since it does point out an important time constraint for the history that otherwise might be forgotten.

³³Once again, this is actually a requirement of any legal history and is only included here for the sake of completeness.

³⁴The value at this point will be: (a) v_1 , if $p_{\text{first}} = p_1$ and $t_4 \leq t_a - 1$; (b) v_2 , if $p_{\text{first}} = p_2$ and $t_4 \leq t_a$; or (c) 0, in all other cases.

³⁵Same conditions as in the previous case determine the actual value.

³⁶The value at this point will be: (a) v_2 , if $t_7 \leq t_a$; or (b) 0, otherwise.

³⁷Same conditions as in the previous case determine the actual value.

History	Value
$E_1 \cdot E_2 \cdot E_3$	0
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^*$	0
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4$	0
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4 \cdot [E_{\text{reiterate}}]^*$	0
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4 \cdot [E_{\text{reiterate}}]^* \cdot E_5$	0, v1, or v2 ³⁴
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4 \cdot [E_{\text{reiterate}}]^* \cdot E_5 \cdot [E_{\text{reiterate}}]^*$	0, v1, or v2 ³⁵
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4 \cdot [E_{\text{reiterate}}]^* \cdot [E_{\text{reiterate}}]^*$	0
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4 \cdot [E_{\text{reiterate}}]^* \cdot [E_{\text{reiterate}}]^* \cdot E_6$	0
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4 \cdot [E_{\text{reiterate}}]^* \cdot [E_{\text{reiterate}}]^* \cdot E_6 \cdot [E_{\text{reiterate}}]^*$	0
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4 \cdot [E_{\text{reiterate}}]^* \cdot [E_{\text{reiterate}}]^* \cdot E_6 \cdot [E_{\text{reiterate}}]^* \cdot E_7$	0
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4 \cdot [E_{\text{reiterate}}]^* \cdot [E_{\text{reiterate}}]^* \cdot E_6 \cdot [E_{\text{reiterate}}]^* \cdot E_7 \cdot [E_{\text{reiterate}}]^*$	0
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4 \cdot [E_{\text{reiterate}}]^* \cdot [E_{\text{reiterate}}]^* \cdot E_6 \cdot [E_{\text{reiterate}}]^* \cdot E_7 \cdot [E_{\text{reiterate}}]^* \cdot E_8$	0 or v2 ³⁶
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4 \cdot [E_{\text{reiterate}}]^* \cdot [E_{\text{reiterate}}]^* \cdot E_6 \cdot [E_{\text{reiterate}}]^* \cdot E_7 \cdot [E_{\text{reiterate}}]^* \cdot E_8 \cdot [E_{\text{reiterate}}]^*$	0 or v2 ³⁷
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4 \cdot [E_{\text{reiterate}}]^* \cdot [E_{\text{reiterate}}]^* \cdot E_6 \cdot [E_{\text{reiterate}}]^* \cdot E_7 \cdot [E_{\text{reiterate}}]^* \cdot [E_{\text{reiterate}}]^*$	0
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4 \cdot [E_{\text{reiterate}}]^* \cdot [E_{\text{reiterate}}]^* \cdot E_6 \cdot [E_{\text{reiterate}}]^* \cdot E_7 \cdot [E_{\text{reiterate}}]^* \cdot [E_{\text{reiterate}}]^* \cdot E_9$	0
$E_1 \cdot E_2 \cdot E_3 \cdot [E_{\text{reiterate}}]^* \cdot E_4 \cdot [E_{\text{reiterate}}]^* \cdot [E_{\text{reiterate}}]^* \cdot E_6 \cdot [E_{\text{reiterate}}]^* \cdot E_7 \cdot [E_{\text{reiterate}}]^* \cdot [E_{\text{reiterate}}]^* \cdot E_9 \cdot [E_{\text{reiterate}}]^*$	0

Figure 4-7: Histories Accepted by LBESA Beginning With $E_1 \cdot E_2 \cdot E_3$

The maximum total value of any history accepted by LBESA is $\max(0, v1, v2)$. Since v1 and v2 are both greater than zero, this is equal to $\max(v1, v2)$. Also, from the initial value density relations, it is known that:

$$v1/t_a + 1 > v2/t_a$$

Therefore,

$$v1 \cdot t_a > v2 \cdot (t_a + 1) \quad (\text{note that } t_a > 0)$$

$$v2 \cdot (t_a + 1) = v2 \cdot t_a + v2 > v2 \cdot t_a \quad (\text{since } v2 > 0)$$

$$v1 \cdot t_a > v2 \cdot (t_a + 1) > v2 \cdot t_a$$

$$v1 > v2$$

Consequently, the maximum total value for any of the histories in the table is $\max(v1, v2) = v1$.

As shown in the first section of this proof, DASA/ND accepts a history with value $(v1+v3)$ starting with these three events, while the maximum value for a history accepted by LBESA is $v1$. Therefore, there exists a case in which DASA/ND accepts a history with greater value than LBESA, and there is no transformation of that history or alternate history dealing with the same phases and scheduling parameters that allows LBESA to obtain an equal or greater value than DASA/ND.

EndOfProof

Since the DASA/ND Scheduling Automaton is equivalent to the DASA Scheduling Automaton when there are no dependency considerations, the result extends to the DASA Scheduling Automaton as well.

4.3.3. Algorithm Tractability

This section examines the computational complexity of the DASA scheduling algorithm. Specifically, the amount of time and space required for the DASA algorithm to select a phase to execute is derived. Of course, the lower the complexity of a computation, the more feasible it is to perform. In general, problems that have exponential complexity are deemed intractable, while those that have a low polynomial complexity are considered tractable.

4.3.3.1. Procedural Version of DASA

It is possible to use the definition of the 'SelectPhase()' function presented in Section 3.2.1.3 to investigate the computational complexity of the algorithm. However, it seems to be somewhat easier to analyze a procedural definition of the function.

Figure 4-8 shows a procedural definition of the DASA scheduling algorithm.

Where possible, the variable names in the procedural definition are taken from the corresponding state components in the DASA Scheduling Automaton.

The language employed for the definition is similar to Algol or Pascal. The control statements (*if-then-else*, *for*, and *while*) may delimit blocks of code and are explicitly terminated (with *endif*, *endfor*, and *endwhile*, respectively) to avoid any ambiguity. The *for* statement is used to step through an ordered list, one entry at a time. The variables in the *for* statement take on the values dictated by the current element in the list. The *exitfor* statement causes control to pass to the statement following the innermost *for* loop enclosing the *exitfor* statement.

```

SelectPhaseProc(PhaseList) {
    : variable declarations
    schedule          Sched, TentSched
    real              TotalTime, TotalValue, CurrentDeadline, DL
    phase             P, NextP, PriorP, CurrentPhase
    ordered list of phase PhaseList, SortedList
    mode              SchedMode, Mode
    6

    : create an initially empty schedule
    Sched = emptyschedule
    : construct the dependency list and determine PVD for each phase
    for P in PhaseList
        if (ExecMode(P) = normal) then
            TotalTime = ExecClock(P)
            TotalValue = Val(P)
            DependencyList(P) = emptylist
            NextP = Owner(ResourceRequested(P))
            SchedMode = normal
            : follow chain of dependencies
            while ((NextP ≠ nullphase) ∧ (SchedMode ≠ abort))
                if (ExecClock(NextP) ≤ AbortClock(NextP)) then
                    : update dependency list and adjust accumulated value and time
                    DependencyList(P) = DependencyList <complete, NextP>
                    TotalTime = TotalTime + ExecClock(NextP)
                    TotalValue = TotalValue + Val(NextP)
                else
                    DependencyList(P) = DependencyList <abort, NextP>
                    TotalTime = TotalTime + AbortClock(NextP)
                    : note: 'TotalValue' remains unchanged
                    SchedMode = abort
                endif
                : advance to next phase in dependency list
                NextP = Owner(ResourceRequested(NextP))
            endwhile
            PotentialValueDensity(P) = TotalValue/TotalTime
        else
            : if aborting phase, there is no value to be gained directly
            PotentialValueDensity(P) = 0
        endif
    endfor
    : form a sorted list of phases according to potential value density
    : (highest PVD first in list; lowest PVD last)
    SortedList = ScrByPVD(PhaseList)
    41

```

Figure 4-8: Procedural Definition of DASA Scheduling Algorithm

The following simple functions are used in the algorithm definition:

Insert(element, orderedlist, key)
 inserts *element* in list *orderedlist* at the position indicated by *key*; if there are already entries in the list with key value *key*, insert *element* before them.

```

; look at each phase in turn
for P in SortedList
    ; if it has any potential value, attempt to add it to schedule
    if (PotentialValueDensity(P) > 0) then
        ; only add completion if it hasn't already been scheduled
        if (<complete, P> ∉ Sched) then
            ; get a copy of the schedule for tentative changes
            TentSched = Sched
            ; tentatively add 'P' and its dependency list to the schedule
            Insert(<complete, P>, TentSched, Deadline(P))
            CurrentDeadline = Deadline(P)
            CurrentPhase = P
            ; tentatively add phases in dependency list to schedule
            for <Mode, PriorP> in DependencyList(P)
                if (<Mode, PriorP> ∈ TentSched) then
                    ; see if the phase is scheduled soon enough
                    DL = Lookup(<Mode, PriorP>, TentSched)
                    if (DL < CurrentDeadline) then
                        ; it is; nothing else to do so exit the loop
                        exitfor
                    else
                        Remove(<Mode, PriorP>, TentSched, DL)
                    endif
                endif
                if (Mode = normal) then
                    CurrentDeadline = Min(CurrentDeadline, Deadline(PriorP))
                else
                    ; 'CurrentDeadline' remains unchanged
                endif
                ; tentatively add phase to schedule
                Insert(<Mode, PriorP>, TentSched, CurrentDeadline)
            endfor
            ; clean up tentative schedule, as required
            examine current simplifications; make less brute force
            ; test the feasibility of the tentative schedule
            if (Feasible(TentSched)) then
                ; incorporate all of the tentative changes into the schedule
                Sched = TentSched
            else
                ; 'Sched' remains unchanged
            endif
        endif
    endif
endfor
; select first phase to execute
return(First(Sched))

```

Figure 4-8: Procedural Definition of DASA Scheduling Algorithm, *continued*

Remove(element, orderedlist, key)

removes *element* from list *orderedlist* at the position indicated by *key*; if *element* is not present at that position in the list, the function takes no action.

- Lookup(*element*, *orderedlist*) returns the key value associated with the first occurrence of *element* in list *orderedlist*.
- First(*orderedlist*) returns the first element in list *orderedlist*.
- SortByPVD(*phaseslist*) returns a list of phases ordered by decreasing PVD; if two or more phases have the same PVD, then the phase or phases with the greatest required execution time (*ExecClock*) appear before any others with the same PVD.
- Feasible(*orderedlist*) returns a boolean value (*true* or *false*) indicating whether the schedule represented by *orderedlist*, an ordered list of mode-phase pairs, constitutes a feasible schedule, as defined previously (by the function *feasible()* in Section 3.2.1.3).
- Min(*x*, *y*) returns the minimum of *x* and *y*.

Briefly, the procedure consists of four stages. First, each phase is examined to determine its potential value density and to construct its dependency list. Second, the phases are sorted and placed into an ordered list ranked by their PVD. Next, a schedule is constructed by attempting to add each phase, along with all of the other phases in its dependency list, to the evolving schedule. If this addition produces a feasible schedule, then the phase is included in the schedule; otherwise, it is not. (Some simplifications of the evolving schedule occur at this point as well.) Finally, after all of the phases have been considered for inclusion in the tentative schedule, the schedule's first element is selected for immediate execution.

The schedule created by the *SelectPhaseProc()* procedure is an ordered list of mode-phase pairs, each placed according to the deadline it must meet. So, for instance, a phase that must meet a deadline at time $t = 1$ will precede a phase that must meet a deadline at time $t = 2$ in the schedule. If more than one phase must meet a single deadline, then the mode-phase pair that was added to the schedule last will be executed first.

Notice that the deadline a mode-phase pair must meet is not necessarily the deadline associated with that phase. In fact, the phase may need to meet an earlier deadline in order to enable another phase to meet its time constraint. Whenever a phase is considered for insertion in the tentative schedule (line 47 of Figure 4-8), it is scheduled to meet its own time constraint. However, all of the mode-phase pairs in its dependency list must execute before it can execute, and, therefore, must precede it in the schedule.

The variable *CurrentDeadline* is used in *SelectPhaseProc()* to keep track of this type of scheduling consideration. Initially, it is set to be the deadline of the phase to be tentatively added to the schedule. Thereafter, any mode-phase pair that has a later time constraint than *CurrentDeadline* is required to meet *CurrentDeadline*. If, however, a mode-phase pair has a tighter deadline than *CurrentDeadline*, then it is scheduled to meet the tighter deadline, and *CurrentDeadline* is advanced to that time since all of the mode-phase pairs left in the dependency list must complete by then.

The major data structures used by *SelectPhaseProc()* are:

1. a Phase Control Block (PCB) for each phase to be scheduled – it contains a phase id, the necessary scheduling parameters (*ExecMode*, *ExecClock*, *AbortClock*, *Deadline*, *Value*), the names of any currently requested or held shared resources, a reference to a dependency list, and a reference to another phase that is used to chain PCBs together to form the *PhaseList*;

2. *PhaseList* is simply a reference to the first phase in the list; subsequent phases in the list are found by following the phase reference field in the PCBs;
3. *SortedList* is simply an ordered list of references to the PCBs;
4. dependency lists are linked lists of mode-phase pairs, each of which refers to a specific PCB;
5. schedules are ordered lists of mode-phase pairs; although many data structures may be sufficient, assume a balanced binary tree is used here³⁸ (for example, a 2-3 tree); then insert, remove, lookup and find minimum operations can all be done in $O(\log N)$ time and $O(N)$ space for a set of N phases.

4.3.3.2. Proof: Procedural Version of DASA Is Polynomial in Space and Time

Given the definition of *SelectPhaseProc()*, it is possible to demonstrate that it requires an amount of space and time that is proportional to a polynomial power of the size of the problem: the number of phases requesting to be scheduled.

Theorem 4: Given N phases to be scheduled using the DASA scheduling algorithm, show that *SelectPhaseProc()* will determine the first phase to execute in $O(N^2 \log N)$ time.

Proof. To determine the time required by *SelectPhaseProc()*, examine the amount of time required for each of its component steps:

1. create an initially empty schedule (lines 9-10): $O(1)$, this requires constant time for virtually any list structure.
2. construct the dependency list and determine PVD for each phase (lines 11-40): $O(N^2)$, since:
 - a. the *for* loop begun at line 12 is executed N times, once for each phase;
 - b. if the *ExecMode* of the phase is not *normal*, then the loop body takes $O(1)$ time to execute (it is a single assignment statement, lines 37-38); however, if the *ExecMode* is *normal*, then loop body takes $O(N)$ to execute since:
 - i. lines 14-18 require $O(1)$ time;
 - ii. because there are no deadlocks, there can be no circular dependency lists; therefore, the *while* loop at line 20 will be executed less than N times, and each time lines 21-33 require $O(1)$ time; hence the entire *while* loop requires $O(N)$ time to execute in the worst case;
 - iii. line 35 requires $O(1)$ time;
3. form a sorted list of phases according to potential value density (lines 41-43): $O(N \log N)$ if any of a number of standard sorting algorithms are used (for example, heap sort);
4. tentatively add each phase in turn to the schedule (lines 44-87): $O(N^2 \log N)$, since:
 - a. the body of the *for* loop at line 45 will be executed N times, once for each phase;
 - b. the loop body takes $O(1)$ time to execute if the phase's PVD is less than or equal to zero or if the completion of the phase has already been scheduled; otherwise, it requires $O(N \log N)$ because:
 - i. copying the schedule (lines 50-51) can be done in $O(N)$ time in a straightforward manner;

³⁸Given a specific type of application, experience may indicate that there are better data structures for schedules. For example, if there are typically only a few phases ready to execute, then a simple linear, linked list may be sufficient. The tree structure was selected for generality and because it will accommodate large numbers of phases and dependencies gracefully.

- ii. inserting the completion of the phase into the schedule (lines 52-53) can be done in $O(\log N)$ time since there are at most $2N$ mode-phase pairs in the schedule (corresponding to an abort and a normal completion for each of the N phases);
 - iii. setting up some variables for bookkeeping (lines 54-55) requires $O(1)$ time;
 - iv. the *for* loop (lines 56-75) requires $O(N \log N)$ time since the loop will be executed fewer than N times and each execution will require $O(\log N)$ time to perform insert, remove, and lookup operations on the tentative schedule;
 - v. testing the feasibility of the tentative schedule (lines 78-79) requires $O(N \log N)$ time since it can be done by looking up each of the scheduled mode-phase pairs in order, summing execution requirements, and comparing those requirements to the actual available time; this requires N lookups, each requiring $O(\log N)$ time;
 - vi. incorporating all of the tentative changes into the schedule (lines 80-81) require $O(N)$ time; this can be done by copying the N nodes that comprise the tentative schedule over the existing schedule entries;
5. select first phase to execute (lines 88-89): $O(\log N)$ time

Therefore, the overall time to execute *SelectPhaseProc()* is $O(N^2 \log N)$.

EndOfProof

The preceding proof uses straightforward data structures and algorithms. An actual implementation may be able to improve on these. For instance, a number of the calculations performed to compute the PVD for each phase could be avoided if it was noted that the phase and its dependency list had not changed since the last execution of *SelectPhaseProc()*. Such an optimization trades storage for speed. Other similar optimizations may bring additional savings.

Theorem 5: Given N phases to be scheduled using the DASA scheduling algorithm, show that *SelectPhaseProc()* will determine the first phase to execute using $O(N^2)$ space.

Proof. The space required for *SelectPhaseProc()* consists of:

- 1. a PCB for each phase to be scheduled — this requires $O(N)$ space;
- 2. two schedules, *Sched* and *TentSched*, each of which is a balanced binary tree with at most $2N$ nodes — this requires $O(N)$ space;
- 3. space for *SortByPVD()* to sort the phases (actually, it will sort a set of keys that refer to individual PCBs) — this requires $O(N)$ space;
- 4. space for each phase's *DependencyList* — this requires $O(N)$ space for each phase in the worst case, thereby requiring $O(N^2)$ space overall in the worst case³⁹;
- 5. various scratch variables — this requires $O(1)$ space.

Putting these requirements together, it is seen that, in the worst case, *SelectPhaseProc()* may require $O(N^2)$ space.

³⁹This would truly be unusual. In order to have very long dependency lists for each phase, the system would have to be nearly deadlocked and every phase would have to be close enough to completing its normal execution that it would take longer to abort than to let it complete normally.

Notice that there is no mention of the storage required to track the ownership and state of each of the shared resources in the system. This is ignored because it is information that is always maintained by the system for any resource management or scheduling algorithm. No additional cost is imposed by the DASA algorithm.

EndOfProof

4.4. Notes on Algorithm

The proofs presented in this chapter have allowed the behavior of the DASA scheduling algorithm to be witnessed under specific circumstances, providing more understanding of the algorithm. This, coupled with the algorithm's formal definition, may suggest situations where DASA may exhibit unusual or unexpected behavior.

Each of the following sections discusses one such situation and the attendant algorithm behavior. Where appropriate, methods for handling the situation are also mentioned.

4.4.1. Unbounded Value Density Growth

While value density and potential value density are appealing because they allow the application to make the best use of the processor time consumed by each phase, they also display an interesting behavior when the required computation time to complete a phase approaches zero: the value density, which is value divided by required computation time, becomes unboundedly large.

This can have some unexpected effects, since — given a sufficiently short required computation time — DASA will favor executing a phase with a very low actual value over a phase with an extremely high actual value that requires more time. In fact, this is arguably the proper decision to make, given that the scheduler's objective is to maximize total value to the application, not to execute the phase with the greatest value.

When assigning values to phases, an application designer may wish to insure that, under any circumstances, a given phase will be selected for execution over another phase. In order to do this, the designer must insure that the value density of the desired phase is always the greater of the two value densities. However, if the value density can grow unboundedly large, then, in general, there is no way to guarantee that the value density of one phase will always be greater than that of another phase.

A few facts mitigate this problem, though. For one thing, required computation time will never reach zero because if it did the phase would be done and would not be involved in scheduling decisions. Therefore, there is a limit on how small the required computation time can be. Hence there is also a bound on how large a value density can grow. The application designer can use this bound to assign values appropriately.

If that bound is deemed to be too large, then a smaller bound can be imposed by specifying a minimum amount of computation time that may be requested for completing a phase. If a required computation time parameter should ever be smaller than this minimum, then the minimum value should be used in its place when applying the DASA scheduling algorithm.

Evaluating the value density associated with a phase only once, at the time of the phase's initiation, would also have the effect of avoiding the practically unbounded growth of value densities. The basic information encoded into the value density metric would remain the same and would be captured effectively. However, the benefit that arises from evaluating the value density for each scheduling decision would be lost — that is, there would no longer be a rising value density to indicate that for a relatively small investment of processor cycles, a large return in application value could be realized.

4.4.2. Idle Intervals During Overload

DASA is not optimal; it is a heuristic that does well according to important metrics for the class of real-time supervisory control applications. However, there are overload situations where it can be less effective than other scheduling algorithms.

DASA constructs a schedule by successively adding activities that have the highest PVDs. In this way, each time an activity, along with any other activities on which it depends, is added to the tentative schedule, DASA is getting the greatest amount of value for the processing cycles that are then reserved for those activities. (If any other activity could yield more value for those processor cycles, it would — by definition — have a higher PVD. But all of the activities with a higher PVD that can be feasibly scheduled have already been added to the tentative schedule.)

LBESA adds activities to a schedule according to the nearness of their deadlines; and, in case of an overload, it sheds the activities with the lowest PVDs until a feasible activity is obtained. As shown in Section 4.3.2.4, LBESA may shed some activities that can be included in a schedule. This can result in LBESA utilizing fewer processor cycles than DASA in a given situation.

The factors discussed in the previous paragraphs can collectively yield a situation where LBESA can produce a schedule representing a higher value to an application than can DASA. For instance, consider an application consisting of three activities, each of which has only a single phase. The phases are designated p_1 , p_2 , and p_3 , respectively. Furthermore, assume that at time $t = 0$ the following conditions hold (using the notation for the scheduling automata):

$$Deadline(p_1) < Deadline(p_2) < Deadline(p_3)$$

$$PVD(p_2) > PVD(p_1) > PVD(p_3)$$

$$ExecClock(p_1) \leq Deadline(p_1)$$

$$ExecClock(p_2) > Deadline(p_2)$$

$$ExecClock(p_3) \leq Deadline(p_3)$$

$$ExecClock(p_1) + ExecClock(p_3) > Deadline(p_3)$$

Among other things, these conditions indicate that phase p_2 cannot be completed by its deadline, even if no other phases are executed. Also, either phase p_1 or phase p_3 , but not both, can meet their deadlines.

When DASA is presented with this situation, it constructs a tentative schedule by examining each phase in order of decreasing PVD. Consequently, it will:

1. add phase p_2 to the (initially empty) tentative schedule, determine that the schedule is not feasible, and shed phase p_2
2. add phase p_1 to the tentative schedule and determine that the schedule is feasible
3. add phase p_3 to the tentative schedule, determine that the schedule is not feasible, and shed phase p_3

This results in a tentative schedule that contains only phase p_1 .

When LBESA is presented with this situation, it constructs a tentative schedule by examining each phase in order of increasing deadline. Consequently, it will:

1. add phase p_1 to the (initially empty) tentative schedule and determine that the schedule is feasible
2. add phase p_2 to the tentative schedule, determine that the schedule is not feasible, shed phase p_1 , determine that the schedule is still not feasible, and shed phase p_2 (leaving an empty tentative schedule)
3. add phase p_3 to the tentative schedule and determine that the schedule is feasible

This results in a tentative schedule that contains only phase p_3 .

Comparing the results, whenever the value associated with phase p_3 is greater than that associated with phase p_1 , then LBESA will accrue a higher value than DASA. In addition, this implies:

$$\begin{aligned} Value(p_3) &> Value(p_1) \\ &\rightarrow ExecClock(p_3) \times PVD(p_3) > ExecClock(p_1) \times PVD(p_1) \\ &\quad \frac{ExecClock(p_3)}{ExecClock(p_1)} \times PVD(p_3) > PVD(p_1) \{ > PVD(p_3), \text{ from above } \} \\ &\rightarrow ExecClock(p_3) > ExecClock(p_1) \end{aligned}$$

For the DASA-produced schedule, the processor is idle for $Deadline(p_3) - ExecClock(p_1)$ units of time, while for the LBESA-produced schedule, the processor is idle for $Deadline(p_3) - ExecClock(p_3)$ units of time. Therefore, the schedule produced by DASA has more idle time than the one produced by LBESA —

even though there is an overload and two of three phases that were known to the scheduler were shed. Consequently, by executing an activity with a lower value density for a long enough time, while the DASA scheduler is forced to leave the processor idling, LBESA can accrue a greater value than DASA for an application.

4.4.3. Cleverness and System Dynamics

The applications of interest for this research are by nature dynamic. A scheduler must be able to react dynamically in order to produce effective schedules for these applications.

Yet there is a balance to be struck. The more information that is used to make scheduling decisions, the better-informed the decisions are. This typically results in better scheduling decisions. On the other hand, each decision is made based on the best information available *at the time of the decision*. At any point thereafter, circumstances may change — a new request may be made for a shared resource or new activities may arrive to be scheduled — demanding that new scheduling decisions be made, possibly resulting in undoing some previously accomplished work.

Intuitively, the more dynamic and unpredictable an application is, the less appropriate clever (read "time-consuming") scheduling schemes are. The actual dividing line for this decision is not clear in general. The simulations in the following chapter demonstrate DASA's performance in various situations and take into account the amount of time required to make scheduling decisions. In fact, the simulator could be used to determine the effectiveness of the DASA scheduling algorithm compared to another algorithm for any application.

Chapter 5

Simulation Results

The formal analysis presented in the previous chapter shows that the DASA algorithm possesses some desirable properties. These properties were demonstrated by comparing the behavior of DASA to other known algorithms. However, this analysis did not evaluate the use of the DASA algorithm in particular situations, nor did it quantify the gains that could be realized by using DASA to schedule specific workloads. Simulations were employed to examine these issues, and that work is described in this chapter.

Section 5.1 discusses the design and implementation of the simulator used to evaluate DASA and other scheduling algorithms. Section 5.2 presents results generated with the simulator to evaluate the performance of the DASA scheduling algorithm. Finally, Section 5.3 hypothetically characterizes two real-time applications and outlines how the simulation results can be applied to estimate how well DASA would schedule these applications.

5.1. Simulator Design and Implementation

The first part of this section outlines the set of requirements that the simulator had to meet. The other parts describe the design that was adopted and discuss significant implementation issues.

5.1.1. Requirements

Fundamentally, the simulator must allow DASA to schedule a variety of workloads. In fact, there are a number of ways in which this may be accomplished. Therefore, to guide the simulator development, the following general requirements were adopted:

1. support a variety of workloads conforming to the computational model presented earlier — that is, the simulated workload represents a real-time supervisory control application, which is composed of a number of activities, each of which may have one or more computational phases. The activities may share resources as outlined previously in this work. And all of the assumptions concerning the information that is available to the scheduler, such as the amount of computation time to complete each phase, continue to hold. The set of applications that can be run must be rich in order to allow a significant range of applications to be explored.
2. offer standard statistical distributions for use by the application — to examine the behavior of a scheduler under general conditions, it is often convenient to assume that events occur temporally according to a standard statistical distribution, such as a normal or a Poisson distribution.
3. incorporate useful metrics and gather statistics — the metrics are intended to aid in the

evaluation of scheduler performance. For instance, the number of time constraints satisfied, the number not satisfied, and the total application-specific value accrued are all straightforward examples of useful metrics that the simulator should support.

4. allow evaluation of multiple scheduling algorithms and resource management policies — the primary objective of the simulations is to compare the performance of DASA with that of other algorithms. Therefore, the simulator must accommodate a set of well-known scheduling algorithms, including priority, deadline, and best-effort schedulers. In addition, since DASA also makes all of the shared resource management decisions, the simulator must provide several alternative resource management policies, including FIFO and deadline queueing for shared resources that are not available.
5. provide a trace of the scheduling events and decisions made during a simulation — this information is useful for at least three reasons: (1) it allows a detailed inspection of scheduler behavior to identify specific beneficial or detrimental decisions, (2) it makes available raw data that may be processed to generate other meaningful statistics for any specific scheduler, and (3) during the initial implementation or subsequent modification of a scheduling algorithm, the event trace can be examined by hand or by machine to demonstrate correct behavior.
6. possess the flexibility to adapt to changing requirements or to augment the initial capabilities of the simulator — since the simulator is used to examine algorithms under a wide range of circumstances and the appropriate metrics are not necessarily known in advance, flexibility is desirable. In addition, if the simulator is to be useful over time, it will have to be able to accommodate new algorithms that will be developed, which may or may not resemble those that already exist. By choosing internal interfaces carefully, this is not too demanding a requirement.

The simulator developed meets all of these requirements, as explained in the following sections.

5.1.2. Design

The simulator design compartmentalized major functions so that different workloads and scheduling algorithms could be accommodated. As shown in Figure 5-1, the simulator features several independent parts:

1. a set of shared resources,
2. a set of application activities, each potentially comprising a sequence of computational phases governed by a time-value function, that may access the shared resources,
3. a Simulated Operating System, including an *Integrated Scheduler* — that is, a scheduler that not only manages processor cycles, but also controls access to all shared resources, and
4. an Activity Generator that adds new activities to the application.

5.1.2.1. Activities and the Activity Generator

The Activity Generator initiates the application by creating the first activity or activities. It may subsequently create others while the simulated application is executing.

The activities comprising an application may either be chosen from a library of existing activities or they may be written specifically for the application. In this way, any activity can be included in an application.

In addition, customized Activity Generators can be written to initiate these activities at any time, obeying

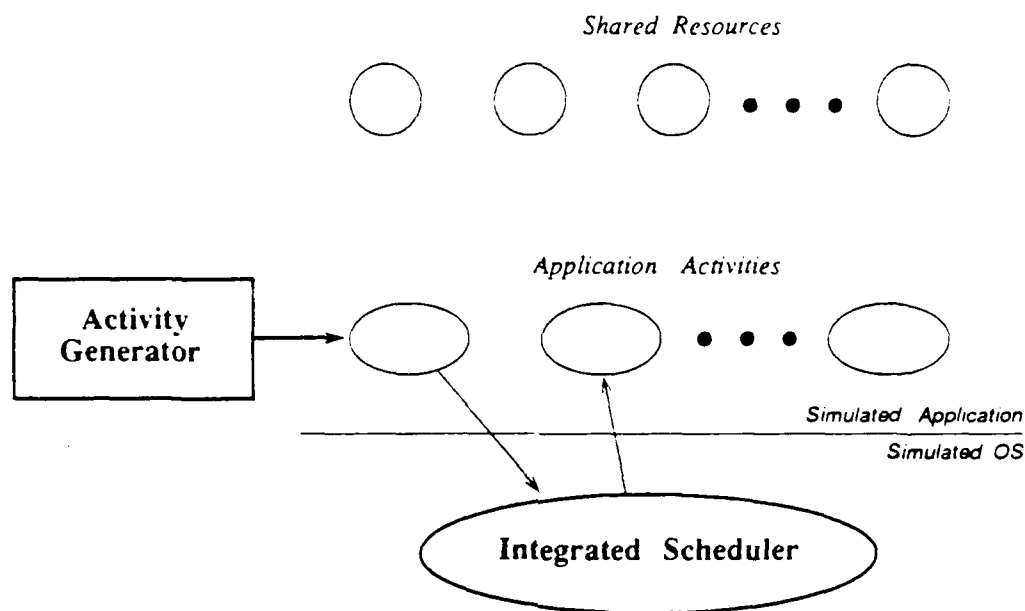


Figure 5-1: Logical Structure of Simulator

any constraints imposed by the actual application being simulated. Therefore, this scheme will support arbitrary workloads.

The activities may mimic computations performed by real applications or they may consume processor cycles and access shared resources in patterns similar to actual or potential applications.

Whenever necessary, an activity will interact with the Simulated Operating System to acquire specific services. The requests made to the Integrated Scheduler, such as requesting the start of a new computational phase or requesting access to a shared resource, are of particular interest for this research.

5.1.2.2. Integrated Scheduler

The interface to the Integrated Scheduler conforms to the interface described in Section 2.3.2 for the General Scheduling Automaton Framework, incorporating scheduling events that are concerned with both processor cycle management and shared resource management.

Scheduling algorithms are embodied in Integrated Schedulers, and different scheduling algorithms can be compared by executing the same application using various Integrated Schedulers.

The requests made of the Integrated Scheduler can naturally be divided into two groups: (1) those that deal fundamentally with phase execution (that is, 'request-phase,' 'abort-phase,' 'preempt-phase,' and

'resume-phase') and (2) those that deal fundamentally with resource management (that is, 'request' and 'grant'). Traditionally, these two groups of requests have been handled by two different entities — the scheduler and the resource manager, respectively. The simulator design at the highest (interface) level hides that distinction. Internally, however, for typical scheduling algorithms requests are routed to the scheduler or the resource manager.

On the other hand, DASA is an integrated scheduling algorithm in this sense, and so all of the requests originating from application activities are directed to the DASA scheduling module.

5.1.3. Implementation

Given a design, the implementation of the simulator raises several new issues, including the selection of the tools to build the simulator, the languages to be used, the interface presented to the experimenter, and the structure of the implementation. Some of the more interesting aspects of these issues are discussed in the following paragraphs.

5.1.3.1. Approach: Build from Scratch or Adapt an Existing Simulator

There are several different approaches that may be used to produce the simulator described above, and selecting one of them is the first major implementation issue to be resolved. For example, the simulator may be custom-built from scratch. This approach allows the simulator to be precisely tailored to meet the goals of this investigation. On the other hand, if an existing simulator could be found that is similar in purpose to the desired simulator, then it might be modified to satisfy the present goals. Possibly, this could be done quickly to generate useful results.

In fact, the approach used — writing the simulator using SIMSCRIPT II.5, a programming language intended for simulations — falls between those two extremes. It builds on previous work, while allowing a large degree of customization.

SIMSCRIPT provides a basic framework and a number of useful libraries, including a random number generator and a full complement of probability distributions. Using SIMSCRIPT obviates the need to reimplement and debug these features for a simulator. In addition, SIMSCRIPT provides a programming abstraction called a *process* that is well-suited to model an activity. These processes may control their own (virtual) execution, as well as that of other SIMSCRIPT processes. The code that comprises the Integrated Scheduler is executed by processes when they initiate a scheduling event. The scheduling algorithm dictates the resulting outcome: either the executing process will continue to run or it will block itself while unblocking its successor. Programming constructs exist to consume (virtual) execution time, and SIMSCRIPT manages the advancement of virtual time.

SIMSCRIPT also supports a programming abstraction called a *resource* to embody shared resources. However, this abstraction, although providing the services of a typical resource manager, was not flexible for the purposes of this work, where the resource management decisions are more closely tied to scheduling decisions. Therefore, some of the resource features of SIMSCRIPT were superseded for these simulations.

The use of a simulation programming language provided sufficient freedom so that the Integrated Scheduler could be implemented in the modular fashion described in the design discussion. If an existing simulator had been chosen as the vehicle for this work rather than a simulation language, then the organizational structure imposed by the simulator might have precluded this possibility.

5.1.3.2. Source of DASA Implementation

The version of the DASA scheduling algorithm that was included in the simulator was adapted from the procedural version of the algorithm presented in Section 4.3.3.1. A procedural version had to be used, since SIMSCRIPT is a procedural language. A straightforward translation converted the Section 4.3.3.1 version into a SIMSCRIPT version.

5.1.3.3. Single Scheduler for Simulation

The simulator uses only a single scheduling algorithm (and associated resource queueing discipline) for a given simulation run. The simulator allows the arrival of new activities and phases to be regenerated exactly for specified simulation runs. Therefore, comparing two scheduling algorithms requires two different simulation runs, one for each of the algorithms. Both runs present identical input to the scheduling algorithms. A subsequent examination of the statistical metrics and the scheduler performance for each run can then reveal which algorithm was more effective in the simulated situation.

5.1.3.4. Simulator Display Messages

By default, the simulator displays all of the key information regarding a simulation run to the experimenter. This includes a timestamped message announcing the arrival of each new computational phase that must be scheduled, along with its time constraint, required execution time, value, the number and identity of the shared resources that it will require, and the time interval between each pair of shared resource acquisitions (in terms of actual execution time, not real time). Notice that although the simulator prints information about shared resource needs of a phase at its outset, this information is not available to the scheduling algorithms when the phase is initially presented to the scheduler. Rather, each new resource request is made by the phase at the moment the resource is needed. Only at that point is the scheduler made aware of the need for that particular resource. The information about all of a phase's resource requirements is printed out when the phase initially arrives only as a minor user convenience — it allows all of the requirements information for the phase to be presented together in one place.

Other time-stamped messages are displayed to the experimenter each time a resource is requested or granted or a phase is preempted, resumed, or aborted.

Additionally, a simulation profile is printed that identifies the scheduling algorithm and resource queueing discipline employed, the number of shared resources available, and other workload specific statistics, such as the average interarrival time between phases or the average required execution time for each type of phase.

Finally, a statistical summary of the simulation is displayed at the conclusion of the run. It prints general

statistics including the total number of phases, the number that met their time constraints, the total value represented by all of the phases⁴⁰, and the value actually accrued by the scheduler during the simulation. Other statistics that are of interest for a specific scheduler or workload can also be displayed at the conclusion of the simulation.

All of the messages displayed to the experimenter can be redirected to a file to record the simulation results for later analysis. In this case, the experimenter is offered a summary of the simulation in addition to the log file.

5.1.3.5. Modifications

There are a number of modifications that may be made to the existing simulator, and these modifications can be divided into two groups. First, there are the changes that the simulator was designed to accommodate, for instance, the addition of a new scheduling algorithm or a new resource queueing discipline. Second, there are changes that may be anticipated, but were not specifically provided for in the simulator. Extending the simulator to handle multiprocessor scheduling is an example of the latter type of change.

Provisions have been made to facilitate the anticipated modifications of adding new scheduling and resource queueing policies. To add a new policy, a set of routines must be written, one routine to handle each scheduling event. These routines are named according to an existing convention. The name of the policy is added to the menu of policies available to the experimenter. And finally, the new routines are compiled and linked with the existing simulator.

Since the information required or the data structures used by different scheduling policies may vary significantly, new data fields and structures may be associated with each activity or computational phase. Once again, a naming convention has been adopted for labeling these fields and structures to avoid conflicts with existing fields and structures.

The simulator has been structured carefully so that modifications that could not be anticipated precisely can be handled gracefully. There is no single point in the simulator where all statistics may be gathered and processed. As new statistics are defined, it is likely that at least some of them will have to be inserted in code at locations determined strictly by the scheduling algorithm being examined.

Preparations have been made for some other potential modifications. Some data structures have been defined to be more general than necessary for the purpose at hand. For instance, the number of application processors that are being scheduled is a variable and there is an array containing the relevant state for each of the currently executing activities. Of course there is only one executing activity under the model being investigated by this work. However, in the future the simulator framework may be able to accommodate

⁴⁰Notice that it may not be possible to attain this value, even with complete knowledge of the phases and their requirements. Attaining this total value may be impossible due to insufficient processing cycles or resource availability for some portion of the simulation. It does serve as a clear upper bound on the value that may be obtained by any scheduler.

multiprocessor scheduling. At that time, since many, if not all, of the scheduling algorithms will have to be modified to handle multiprocessor scheduling and use the simulator's data structures in a more general way, it is clear that a great deal of work is required to make this modification to the simulator.

5.2. Evaluation of DASA Decisions

This section evaluates the decisions that DASA makes compared to the decisions made by other scheduling algorithms and resource queueing disciplines. A general, parameterized workload is used to exercise the simulator with varying degrees of processor utilization and varying numbers of shared resources.

5.2.1. Methods of Evaluation

The utility of a scheduling algorithm may be demonstrated in a number of different ways. The following paragraphs deal with four major approaches that correspond to four different workload sources.

5.2.1.1. Execute Existing Applications

Perhaps the most compelling method would be to employ the algorithm in an instrumented, production system and compare the system performance directly to its performance using other algorithms. Using this approach would yield the most direct, relevant information regarding the applicability of the scheduler for a given application.

There are three major problems with this direct approach. First, although it definitely evaluates the performance of the scheduler for a specific application, it is not clear that the information gathered can be applied to any other applications, and if can, under what circumstances. Since this work is addressing a general problem, the ability to make statements that apply to a general class of applications is desirable.

If a wide range of existing applications can be executed directly, this problem can be eliminated and more general results can be derived. However, since many real-time systems today are still custom-designed with customized or proprietary operating systems, finding a large number of real applications that execute under the same operating system may be difficult. Alternatively, modifying the schedulers of several different operating systems may be very difficult logistically.

The second major problem with the direct approach is more specific to the DASA algorithm: the algorithm is significantly different than those that are used in practice today, and it expects that the application will provide the scheduler with more information than is normally the case. (Specifically, the scheduler should be given an estimate of the required computation time needed to execute each new computational phase.) Although this information is often known to application designers and implementers, it is not communicated to the scheduler. As a result, the interface to the scheduler that the application sees is different for the DASA algorithm than for traditional algorithms. This requires that every application used must be altered to provide that additional information to the scheduler, possibly long after the people who knew the information are no longer available or able to provide it.

The final major problem results from the fundamental difference in philosophy between traditional real-time systems and the more dynamic systems that could employ a scheduling algorithm such as DASA. Traditionally, many real-time systems are designed to be quite specialized with minimal overhead resulting from operating system functions. In fact, the designers of these systems attempt to eliminate operating system functions insofar as possible, often either reducing it to the point where it is more correctly termed an executive or eliminating it entirely by having the application perform all required functions.

In such real-time systems, not only are operating system functions limited, but the information supplied to the operating system is minimal. For example, the computational and timing requirements of a given set of activities may be sufficiently studied so that it is possible to replace a priority scheduler, for example, with a list scheduler or a rate-group scheduler. Neither the list scheduler nor the rate-group scheduler display dynamic behavior — at predetermined times they dispatch predetermined activities. All timing and dependency considerations have already been taken into account by the system designers, and the real-time system is unaware of any of this information⁴¹.

As a result, the implementations of real-time systems traditionally distort the application's structure. For example, often physical processes are modeled as periodic, even if they are not, in order to simplify scheduling and increase system predictability. Or shared data is accessed directly (without using an access control mechanism such as a lock) because the activities have been designed and placed in a sufficiently static schedule that it can be demonstrated that no conflicts can occur.

The philosophy underlying DASA resides at the opposite end of the spectrum: in order to handle dynamic applications today and to effectively accommodate application modifications tomorrow, the system always decides which activities should be run, relying on key information supplied by the application. Rather than changing the application in order to restrict the information passed to the operating system in the hope of reducing the run-time computation performed by the system — rendering the application difficult to adapt along the way — the application is encouraged to provide the system with as much relevant information as possible, thereby potentially allowing the system to make better decisions on behalf of the application.

Unfortunately, this philosophical difference implies that the same application designed and implemented according each philosophy will produce very different code. Once again, this limits the ability to validate the effectiveness of DASA by simply using it to schedule existing applications. It is quite possible, for example, that an existing application employs shared memory but, as mentioned above, never issues any requests for access to the shared resource because an appropriately restrictive schedule makes it unnecessary. It is extremely unlikely that DASA could demonstrate improved performance under such constraints.

⁴¹One of the most unfortunate aspects of such systems becomes evident when they must be modified — perhaps to implement a new function or to add an improved device. Then all of the timing and dependency analyses must be performed again. In fact, modifying such systems may cost nearly as much as the original implementation.

5.2.1.2. Modifying or Reimplementing Existing Applications

The preceding discussion emphasizes the difficulties involved in using existing applications directly to evaluate DASA.

Two of the problems mentioned above — DASA requiring more information than is traditionally supplied to a scheduler and implementations that hide application structure and information from the operating system — can be addressed by modifying existing implementations or by reimplementing them. The new, resulting implementations could then be executed using several different schedulers to evaluate the relative effectiveness of each scheduling algorithm. However, in order to justify any results gained by this approach, the new implementations would have to be verified in some manner. Specifically, they would have to be demonstrably equivalent to the original implementations in all important respects. For real applications, which are often large and complex, this vague-sounding requirement could be arbitrarily difficult to satisfy.

5.2.1.3. Modeling Existing Applications

Creating skeletal applications that represent real applications reduces the amount of work required to produce each application, but complicates the problem of proving that an abstracted application corresponds to the real application in all important ways since, by definition, some details of the application will have been discarded. Justifying that the selection of which details should be retained and which should be eliminated or how all or part of the application should be modeled is once again a vague requirement that would have to be addressed in an *ad hoc* manner for each application in all likelihood.

As with each of the preceding approaches to providing a workload to use to evaluate the DASA scheduling algorithm, this method is only capable of providing information concerning the specific workloads used. There is no guarantee that those applications are representative of real-time supervisory control applications in general, and these limitations must be addressed.

5.2.1.4. Simulating the Execution of a Parameterized Application

The final potential approach to evaluate DASA, and the one actually used, employs a parameterized application or set of applications. The execution of these applications can then be simulated under various scheduling algorithms and performance measured. By selecting useful parameters and varying them over ranges of values more general results can be obtained from this workload than could be drawn from a specific set of applications.

Furthermore, the simulator built for this evaluation can be given an arbitrary application (workload). This allows an experimenter to model a potential application with any desired amount of detail, simulate the application's execution using various schedulers, and decide whether the application can benefit from the use of the DASA scheduling algorithm.

Short of building an application model for execution on the simulator, useful information is still available to allow people with real-time applications to decide if DASA may be of interest to them. The simulation

results that follow span a significant portion of the space of real-time supervisory control applications, based on the variation of a few key metrics. If necessary, additional simulations could be performed in the future to extend these results to other regions of the space or to accommodate new metrics. Given the existence of these data, an application designer or implementer can either profile an existing application or create a thumbnail sketch of a new application to determine where the application lies in the supervisory control space and whether any benefit may accrue if the DASA scheduler is used.

This method — simulating the execution of parameterized applications — was chosen to investigate the utility of the DASA scheduling algorithm because of its ability to evaluate the algorithm over a wide range of situations, rather than just a few specific applications. At the same time, it is able to give generally useful information to real-time application designers and implementers for various conditions and then allows them to investigate their application to any desired degree of detail by means of a specific model for their application. This model can be evaluated using the DASA scheduler, as well as a number of other schedulers of general interest. Once again, new schedulers can be added to enrich the simulator if needed or desired.

Thumbnail sketches of real applications that may benefit from the use of the DASA scheduler are presented in Section 5.3.1.

5.2.2. Workload Selection

The workload used to gather the simulation results that follow featured one basic type of activity that was tailored by a number of parameters. In this workload, each activity consisted of only a single phase. The arrival times of the activities could be drawn from any of a number of probability distributions, and the key parameters that define each distribution — such as the mean for a Poisson or an exponential distribution, the mean and standard deviation for a normal distribution, or the minimum and maximum for a uniform distribution — were specified by the experimenter.

The time remaining until a phase's deadline is also drawn from a specified probability distribution and must always be in the future. Once a deadline has been selected, a fraction — drawn from a uniform probability distribution — of the time remaining before the deadline is specified as the required computation time of the phase. Once again, this is always less than or equal to the amount of time remaining until the deadline. Consequently, any such activity executing in a system with no other activities would meet its time constraint. Therefore, any time a time constraint is not met, this is due to the interaction of multiple concurrent activities.

Given the method of generating new activities, their deadlines, and required computation times, it is possible to generate sequences of activities that may not all be completed successfully. This is clear if the parameters specify a condition where the system is overloaded — for instance, if the average required computation time for an activity was more than the average interarrival time between activities. However, even in situations where, on average, there is a significant amount of idle time, there may be transient overload conditions due to the probabilistic nature of the parameter selection.

The values that are accrued by completing an activity's sole phase by its deadline are also taken from a selected probability distribution.

Finally, the number of shared resources is specified by the experimenter. If there are shared resources, then each activity probabilistically determines how many of these resources it will require during the execution of its computational phase. It then selects that number of shared resources randomly. The resource requests are made sequentially with some amount of processing time expended between shared resource requests. The time that passes between each resource request is determined by selecting a fraction of the required computation time for the phase that remains at the time of the previous resource request. For each shared resource, the experimenter may specify the amount of computation time that must be spent to return the resource to a consistent, usable state in the event that the phase is aborted.

Although the resources required by a phase are generated randomly, the actual resource requests are ordered to avoid deadlocks since that is not a primary focus of this work. This is accomplished by associating each resource with a unique key, where all of the keys may be ordered. Requests are then made in increasing order of resource key value. In this way, deadlock cannot occur since it is impossible for any phase to both hold a shared resource that is needed by another phase and to need a shared resource already held by the same phase.

5.2.3. Examination of DASA Behavior

A series of experiments were performed to determine the effectiveness of the DASA scheduling algorithm relative to several other algorithms of interest.

5.2.3.1. Workload Parameters and Metrics

These experiments used the parameterized workload described in the previous section. In this case, activities arrived according to a uniform probability distribution. The time between successive activity arrivals, which is called the *interarrival time*, is between zero and a designated maximum value. This maximum value is varied to examine scheduler behavior under different levels of processor loading.

The deadline for each activity is also drawn from a uniform probability distribution, varying from zero to 200 time units (TUs).

A straightforward load metric is employed for the simulations presented in this section. For these simulations, the expected activity interarrival time is half of the maximum activity interarrival time. Similarly, the expected time remaining until deadline is half of the maximum time remaining until deadline — in this case, 100 TUs. The required computation time for a given activity is expected to be half of the time remaining until its deadline, or 50 TUs. The load metric, then, is simply the expected time required to complete an activity divided by the expected time between successive activity arrivals.

By selecting maximum activity interarrival times from 800 to 50 TUs, the range of processor loads that can be examined extends from 0.125 (fairly light load) to 2.0 (twice as many cycles are required as are available, on average), respectively.

There are two reasons for referring to times in terms of TUs, rather than seconds, milliseconds, or microseconds. First of all, different real-time applications have time constraints that cover a wide range of absolute times. Industrial supervisory control applications typically have time constraints that are measured in seconds or hundreds of milliseconds. Simulators and many military applications may have time constraints that are on the order of tens or hundreds of milliseconds. And lower-level control systems can have even tighter time constraints. By using TUs, this work is not arbitrarily associated with a single class of application. Rather, it seems reasonable to expect that, in the future, these scheduling algorithms can be applied to progressively more demanding real-time applications as processor speeds increase and improved real-time computer architectures are devised.

TUs were also used to allow the results presented here to be reevaluated as technology does change. In particular, the overhead that is incurred by using relatively complex scheduling algorithms can be expressed in terms that reflect the technology of the time, such as the time required to perform a multiplication or division operation or the time required to sort a list. As technology changes, the overhead changes as well.

This can be contrasted with the real-time application being scheduled. Often, the time constraints that must be met are dictated by the application itself — a real world physical process that is subject to the laws of physics for example. Improved computer technology does not affect these time constraints, although it typically affects the application by reducing the amount of processor time that is required to execute any given piece of code. So while the time constraints for a specific physical process remain fixed, the absolute time required to execute both the application and its scheduling algorithm are reduced as technology progresses. This will tend to increase the domain in which complex schedulers may be used in the future.

By expressing both the time constraints and the scheduling overhead in terms of TUs, it is possible to determine what range of time constraints are appropriate for a given scheduling algorithm. To do this, a conversion from real time units (such as milliseconds) to TUs can be computed by noting the time required to perform the basic operations that dominate the scheduling algorithm in question, and therefore are most responsible for its overhead. Using this conversion factor, the application time constraints can be expressed in terms of TUs. Then, a simulation that directly mimics the application in question, including scheduling overhead, can be run, or a more general set of simulations that take scheduling overhead into account can be consulted to determine the applicability of a given scheduling algorithm.

The values associated with the phases varied uniformly from one to ten. A minimum value of zero was not used since that could be interpreted as a worthless process, hence one that need not be scheduled.

A fixed number of shared resources was used for each set of simulations. The results shown in Figures 5-2 through 5-7 correspond to simulations employing zero, one, and five shared resources. Since DASA was the only algorithm that could abort specific phases, the undo times for the shared resources were defined to be (essentially) infinite. In that way, DASA would not schedule aborts and its behavior would be more comparable to that of the other algorithms under consideration.

Three other scheduling algorithms were chosen to compare with DASA: DL, a simple deadline scheduler; SPRI, a static priority scheduler; and LBESA, Locke's Best Effort Scheduling Algorithm.

These algorithms illustrate a number of points. DL and SPRI apply only urgency or only importance information, respectively, while LBESA and DASA consider both types of information. From another point of view, DL represents the simplest type of deadline scheduler — it simply dispatches activities in order of increasing deadline. If there is an overload, rather than shedding some activities, it continues to schedule all activities in deadline order. LBESA provides an advanced load-shedding capability in a deadline-based scheduler. And DASA continues to extend this load-shedding by considering more activities for execution than the other algorithms. Finally, SPRI must be included since it is the algorithm that is actually used by a large number of supervisory control applications.

Shared resource management for each scheduler (except DASA) is handled quite simply: if a requested resource is available, it is immediately allocated to the activity requesting it. Otherwise, the activity is entered in a FIFO (first-in, first-out) queue for that particular shared resource. When a resource is freed at the completion of a computational phase, the first activity entered in its waiting queue is removed from the queue, given access to the shared resource, and made ready to run. The scheduler may subsequently resume its execution. (Notice that while activities are blocked waiting for a shared resource, they are not considered by any scheduling algorithm other than DASA.)

For each combination of maximum activity interarrival time, number of shared resources, and scheduling algorithm, a series of ten simulations were performed. In each simulation, 100 activities were generated and scheduled.

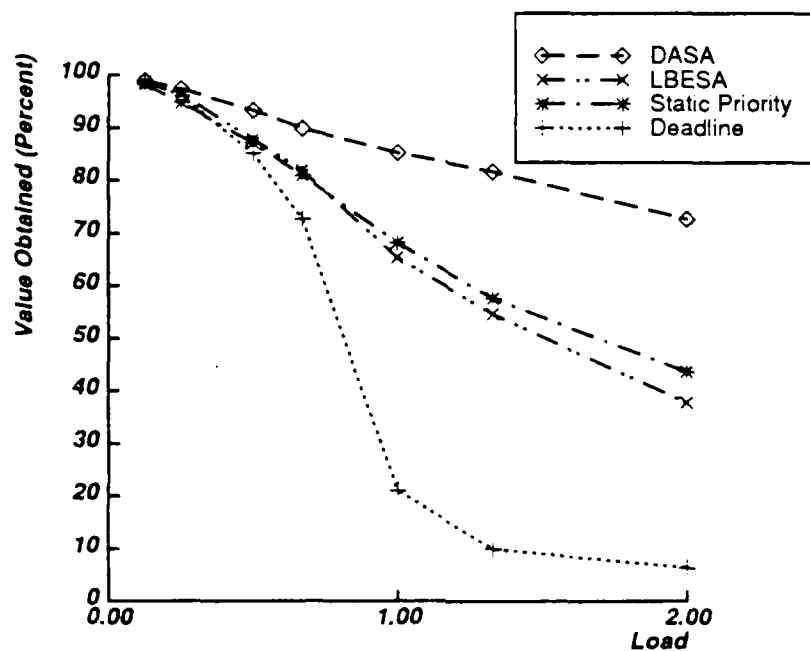
The information gathered by the simulator included a few key metrics: the number of deadlines that were met and the total value represented by all of the activities and that portion of the total value that was actually accrued by the application executing under a given scheduling algorithm. These were reduced to percentages indicating the fraction of deadlines that were met and the fraction of the available value that was obtained.

In addition, the simulator generated an event log that could be examined in order to analyze individual situations and decisions made by various scheduling algorithms.

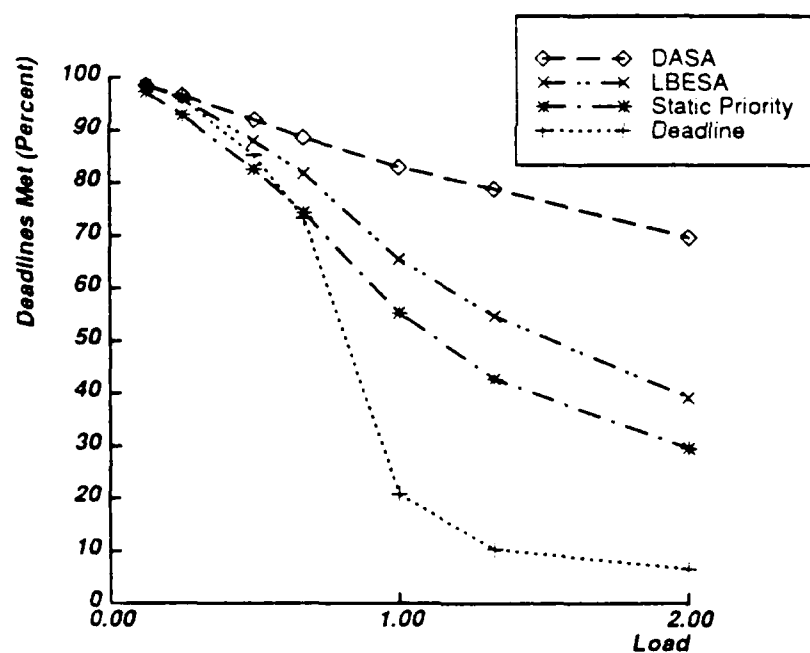
5.2.3.2. Scheduler Performance Analysis

The simulations described in the previous section were performed and the results are shown in Figures 5-2 through 5-7.

Figures 5-2 through 5-4 show the percentage of total available value that was actually obtained and the percentage all deadlines that were actually met when there were zero, one, and five shared resources, respectively, under a variety of processor loads. In these figures, the geometric mean for each scheduling algorithm's performance is plotted as a function of average processor load.



% Value Obtained



% Deadlines Met

Figure 5-2: Average Scheduler Performance with No Shared Resources

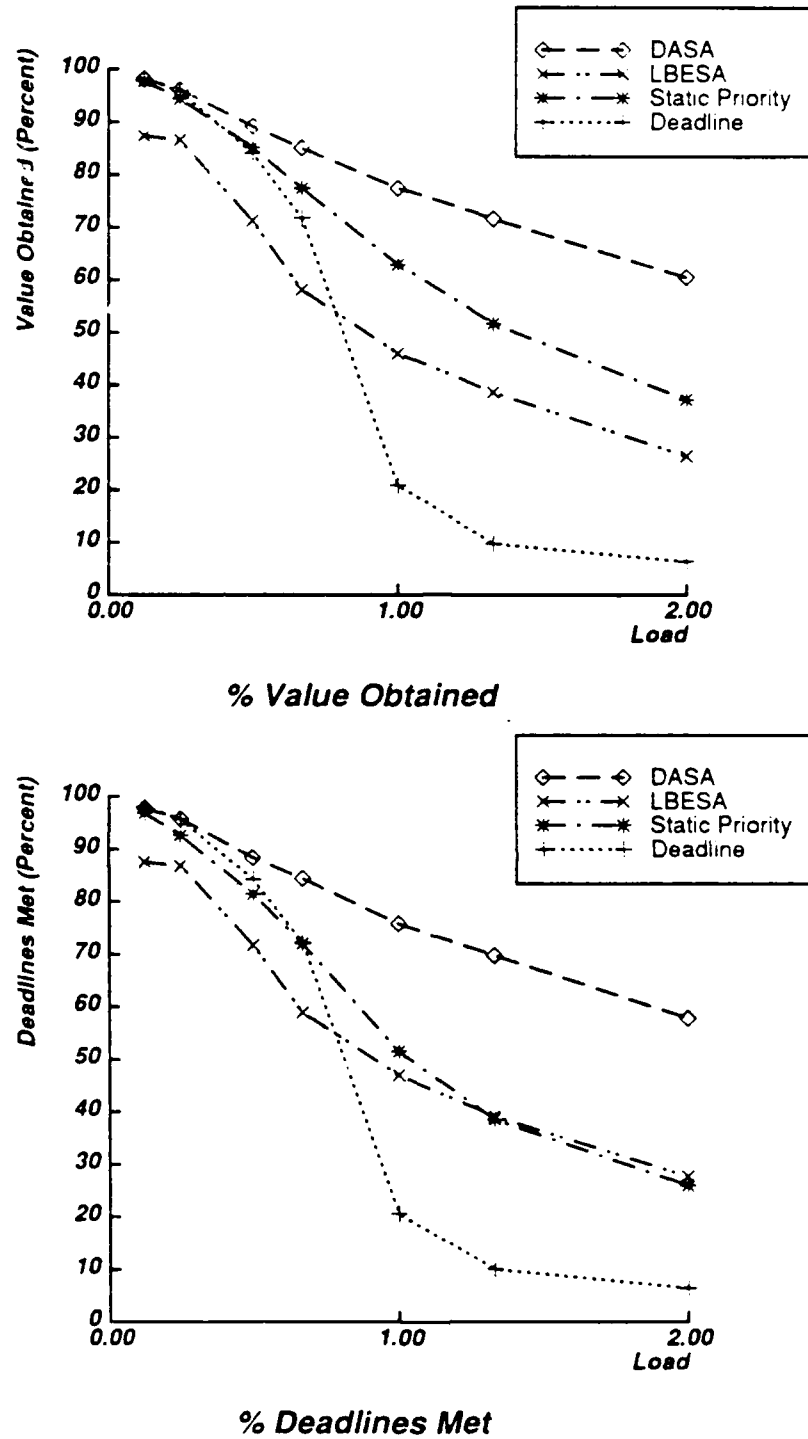
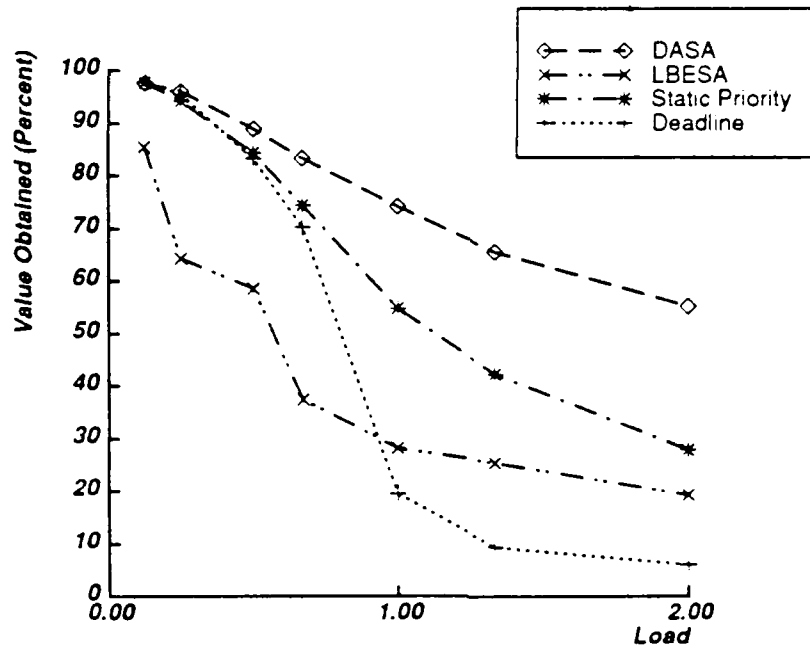
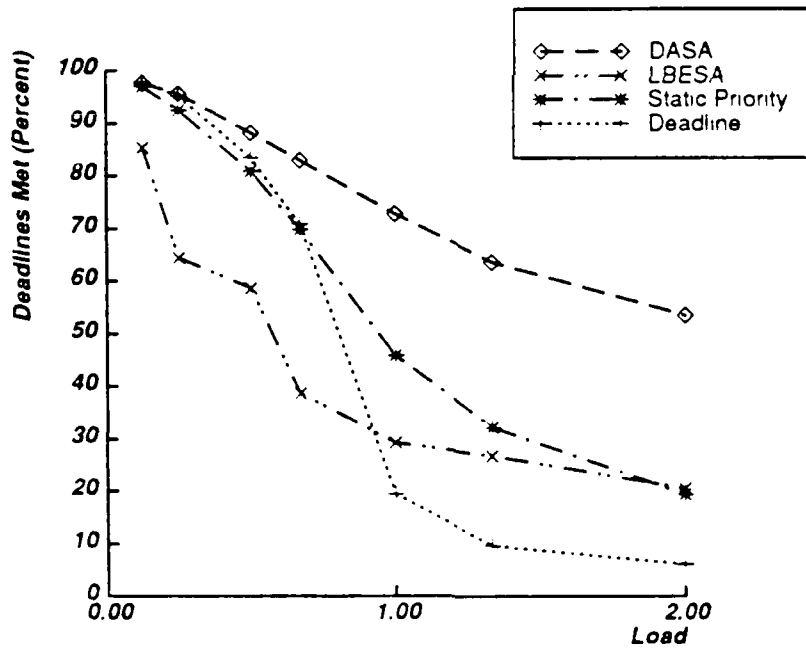


Figure 5-3: Average Scheduler Performance with One Shared Resource



% Value Obtained



% Deadlines Met

Figure 5-4: Average Scheduler Performance with Five Shared Resources

All of the scheduling algorithms perform well under small loads. There are sufficient processing cycles that the exact scheduling algorithm makes little difference. As processor load increases, all of the algorithms become less effective and the differences among them become more apparent. Of course, for loads greater than 1.0, it is impossible to complete all of the activities on time — there are simply not enough processor cycles to satisfy demand. Even for loads that are less than 1.0, there are usually intervals that represent momentary overloads — that is, short intervals of time where it is not possible to complete all of the activities on time — due to the probabilistic nature of the workload. (Consequently, obtaining 100% of the available value or meeting 100% of the deadlines for a simulation is often impossible. However, it serves as an absolute upper bound on the performance of the scheduling algorithms.)

DL drops most rapidly in performance, primarily due to the fact that it does not shed load. This was done intentionally to show an extreme behavior of deadline-based scheduling. LBESA and DASA represent another extreme since they generate schedules that are deadline-ordered and only depart from deadline-ordered schedules when overloads are detected. As shown in Figure 5-2, even when there are sufficient processor cycles, on average, it is still difficult to meet many deadlines using the DL scheduler.

DL does not degrade appreciably with different numbers of shared resources because the overload behavior just described dominates its behavior.

SPRI degrades smoothly as load increases. As more shared resources are added, increasing the interaction of the activities, its performance decreases more rapidly as a function of load.

LBESA exhibits a few noteworthy tendencies. First of all, when there are no shared resources, it typically meets more deadlines than SPRI, while accruing less value. This is partially a consequence of its time-driven orientation compared to the value-driven nature of SPRI. Like SPRI, it also displays graceful degradation as load increases and there are no shared resources.

When there are shared resources, LBESA performs quite differently. It typically performs much worse than any of the other algorithms at relatively low processor loads. This results from a particularly unfortunate interaction between the scheduler and the shared resource manager.

As was pointed out earlier, the actions of the shared resource manager constitute indirect scheduling decisions by blocking activities that had been executing and subsequently determining the order in which they are again made ready (and become visible to the scheduler).

The problem with LBESA and the resource manager arises when an activity requests a shared resource that has previously been allocated to another activity. The requesting activity is then blocked and placed in the FIFO queue for the resource. Later, the resource is allocated to the activity and the activity is added to the ready list for the scheduler. However, if the activity never completes its current phase — either because there is insufficient time to complete it by its deadline or its value density is too low to prevent it from being shed during an overload — it will hold the resource indefinitely. Therefore, all subsequent activities that require access to the resource will fail to meet their deadlines. Of course, this scenario does not result every time an allocated shared resource is requested. But, it does happen occasionally even at low processor loads.

DL and SPRI are not susceptible to this particular interaction because they don't shed load. They eventually execute every activity that arrives. Consequently, any activity that acquires a shared resource will eventually complete execution of its current phase and release the resource. Only algorithms that shed load must be concerned about the fact that activities that are shed may be holding shared resources⁴².

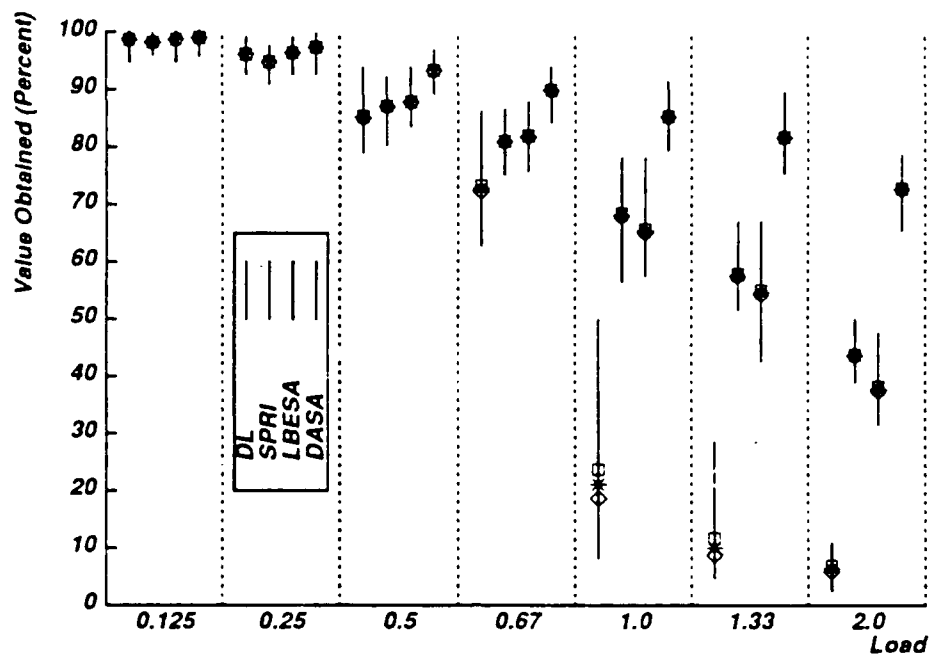
DASA also degrades gracefully as processor load increases, managing to accrue more value and meet more deadlines than any of the other algorithms in these simulations. Even with a load of 2.0, DASA obtains, on average, over 55 percent of the available value — over 25 percent more value than any of the other algorithms.

DASA is not subject to an unfortunate interaction with the shared resource manager since it manages the resources itself. Like LBESA, it will recognize that some activities holding shared resources cannot meet their deadlines and so will not schedule them. However, unlike LBESA, DASA will realize when another activity that can still meet its deadline needs the previously allocated resource and will attempt to execute the activity holding the resource in order to enable continued progress by the application. In this way, processor cycles are not consumed to free allocated resources unless there is an immediate need for the resources. This is in keeping with the general philosophy that the system should always perform the activities that will be most valuable to the system at any time — processor cycles are not expended to free allocated resources unless there is value in doing so.

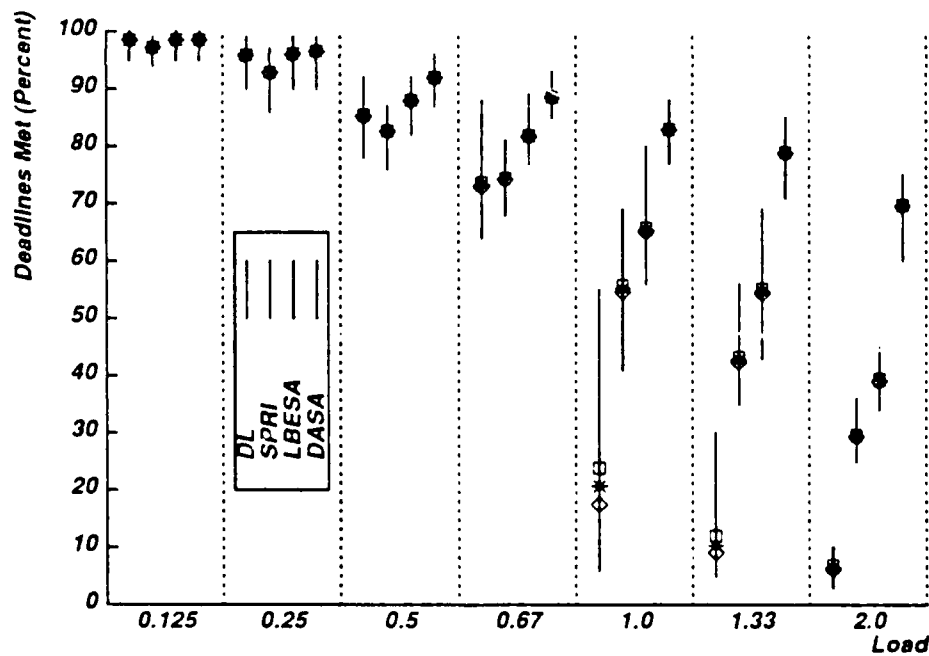
In Section 4.3.2.4, it was shown that DASA could accrue more value than LBESA in during overloads because LBESA could shed some activities unnecessarily. However, it is impossible to prove analytically how often the necessary overload conditions will arise for an application. The simulation results presented in Figure 5-2 show that this effect is quite pronounced under overload conditions — with a 2.0 load, DASA obtains, on average, about 35 percent more value and meets about 30 percent more deadlines than LBESA. (Since there are no shared resources for these simulations, the interaction of LBESA and the shared resource manager has no effect on the simulation results. Under low loads, DASA and LBESA are expected to perform similarly, and under high loads their differences should be due to differences in load shedding.)

Figures 5-5 through 5-7 display more information concerning the simulations described earlier. Where Figures 5-2 through 5-4 plotted the geometric mean for each scheduling algorithm under various loads with differing numbers of shared resources, Figures 5-5 through 5-7 show the range of values obtained and deadlines met in each of these situations. In addition, the arithmetic mean is shown as a box placed along the range; the geometric mean is shown as a star; and the harmonic mean is shown as a diamond. As always for nontrivial data sets, the arithmetic mean is greater than the geometric mean, which is greater than the harmonic mean, for each case. However, the means are often so close that their symbols appear superimposed in the figures.

⁴²LBESA has been modified to execute in the Alpha Operating System. Several adaptations were necessary to use the algorithm in Alpha. The Alpha programming model treats unsatisfied time constraints and communication failures, among others, as exceptions. When an exception is encountered, an associated handler is executed. This handler restores system data structures to acceptable states and offers the application programmer the opportunity to do the same for application data structures and activity state. This offers the opportunity to free shared resources in practice after an unsatisfied time constraint, even though the LBESA model does not address shared resources.

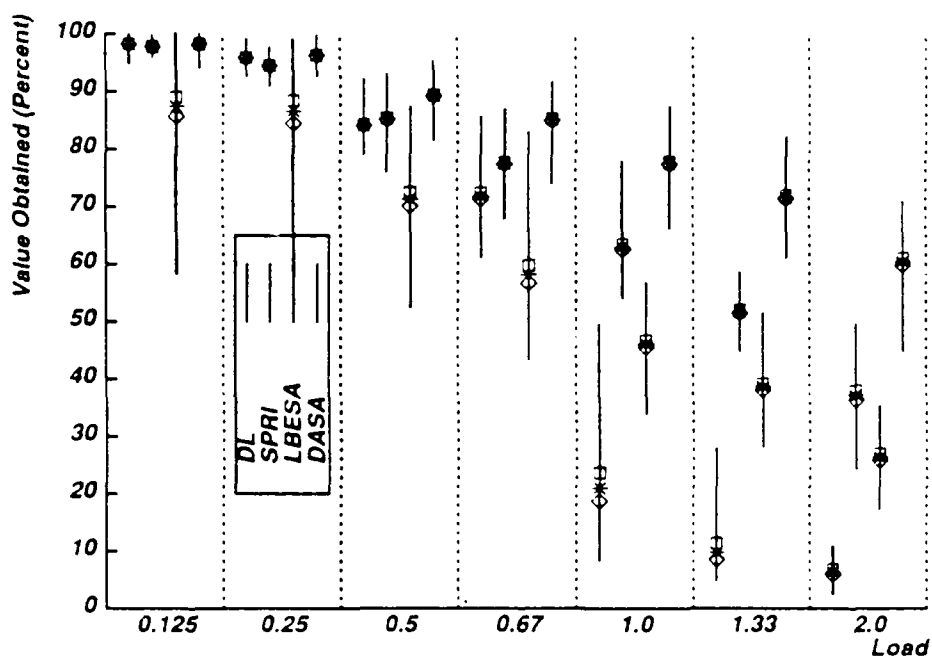


Varying Load, No Resources

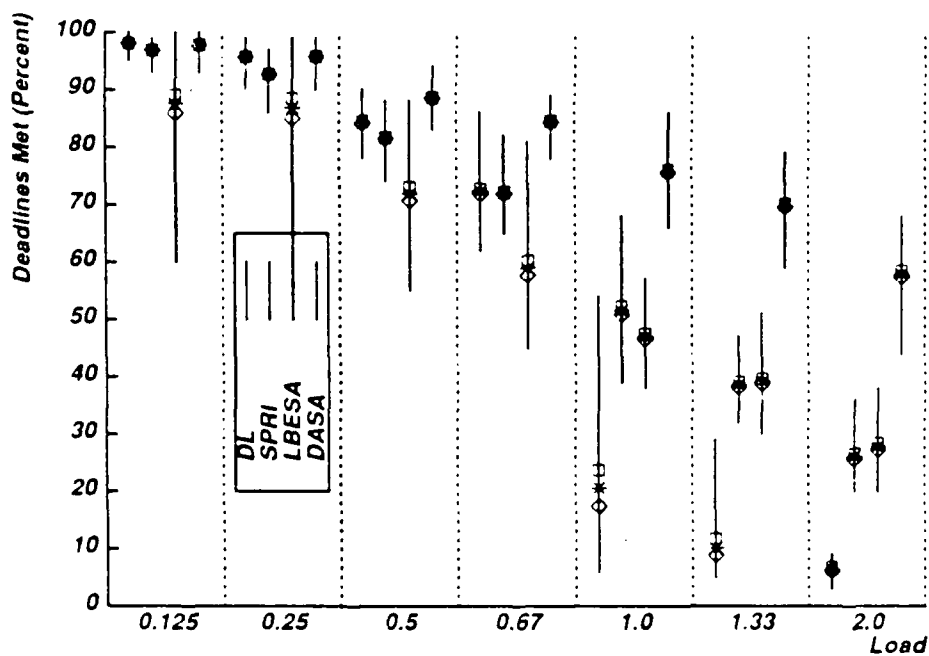


Varying Load, No Resources

Figure 5-5: Scheduler Performance Range with No Shared Resources

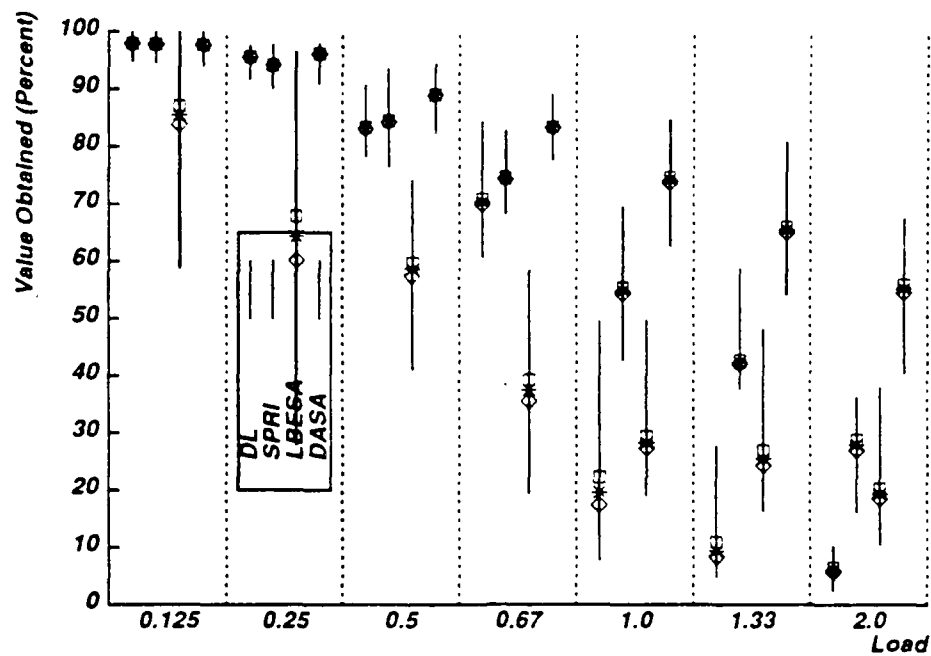


Varying Load, One Shared Resource

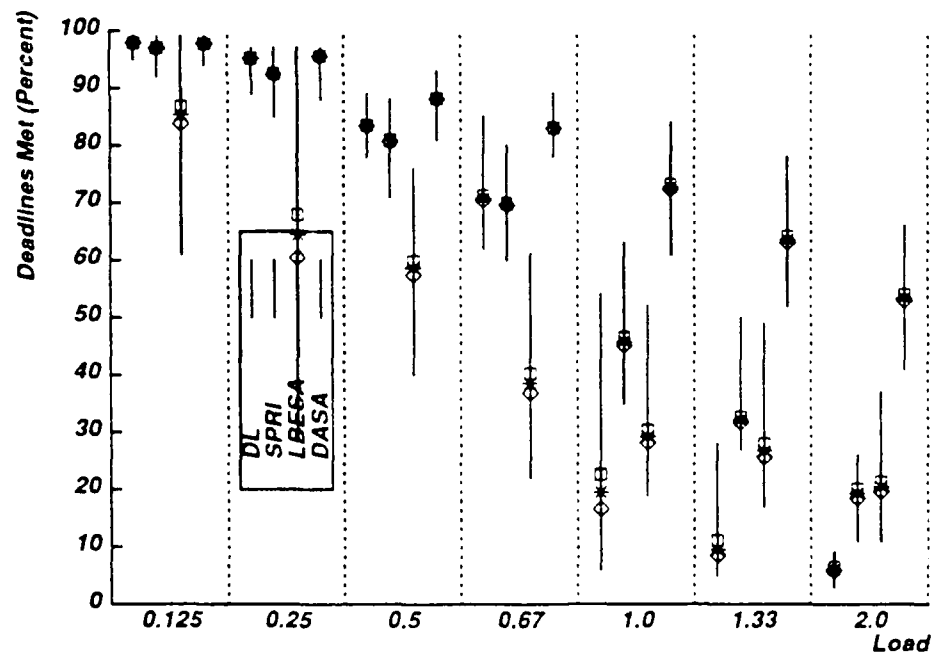


Varying Load, One Shared Resource

Figure 5-6: Scheduler Performance Range with One Shared Resource



Varying Load, Five Shared Resources



Varying Load, Five Shared Resources

Figure 5-7: Scheduler Performance Range with Five Shared Resources

Once again, at low loads with no shared resources, all of the scheduling algorithms perform well, displaying a fairly small variation in performance over multiple simulations. The introduction of shared resources has a marked effect on LBESA's performance, even at low loads — it may perform as well as the others or it may perform much worse (for reasons that were explained earlier in this section).

As load increases, each algorithm displays more variability across multiple simulations. Like LBESA at lower loads, DL's performance falls off sharply as load increases with a great deal of variability.

At higher loads, DASA's performance is superior to the others. In fact, in several cases, the worst performance by DASA for a given set of simulations is superior to the best performance of any of the other algorithms. Furthermore, looking at individual simulations, DASA always outperforms the others at high loads.

5.3. Interpreting Simulation Results for Specific Applications

To complete this chapter, this section looks at a few supervisory control applications for which DASA may be useful. They are indicative of some types of applications that might be of interest, but do not touch on many other possibilities, such as simulators and military platform management.

Each application is outlined briefly, indicating roughly the types of time constraints involved, the processing requirements, and the number and types of shared resources. Application details are provided to explain how supervisory control system requirements are shaped by the physical world. However, the descriptions are necessarily brief since too many details would obscure important information concerning application structure and requirements.

5.3.1. Some Interesting Applications

This section presents thumbnail sketches of two real-world applications that may benefit from the use of the DASA scheduler.

These applications are discussed for two reasons. First of all, they give a flavor of some other applications (in addition to those presented in Chapter 1). The characteristics that make these applications particularly amenable to the use of the DASA scheduling algorithm are noted where appropriate.

Secondly, they offer a chance to use real applications to study how the metrics used for the simulation results can be initially estimated from a knowledge of the application. Of course, the applications are presented briefly here and much better statistics could be gathered more carefully by actual real-time system professionals who were interested in investigating alternate scheduling algorithms.

5.3.1.1. Telephone Switching

The telephone company typically creates a dedicated circuit to handle each telephone call. This circuit is actually composed of a number of shorter circuits that are connected by computer-controlled switches. These switches handle the routing of the call from the originator to the receiver.

Each time a call is initiated, a circuit must be set up to complete the call. At each routing switch along the way, signals must be sent and acknowledged in a moderately short period of time — often on the order of a second.

Because there is no way to associate a priority with a call, it is generally impossible to distinguish urgent calls from less important calls. Therefore, a certain portion of the circuit capacity of the phone company is often held in reserve, even during periods of peak demand, in order to service critical calls in case of an emergency. As a result, this capacity is unavailable for general service and is wasted (in a sense) when there are no emergency calls.

This application could almost certainly benefit by using a scheduling algorithm such as DASA. For instance, the application could be restructured so that each call had an associated priority. As indicated earlier, each call also has a series of time constraints that must be met in order to properly control the switches needed to complete the call. So the simple time-value functions used in this research can be applied directly in this application to capture both the importance and the urgency of each call in the system. So an activity can be assigned to handle each call.

The shared resources in the system are the circuits and switch connections. To complete a call a number of these shared resources must be acquired. At the completion of the call, they may be released.

In addition, no circuits must be reserved exclusively for emergency calls. Therefore, the overall capacity for telephone calls available to telephone company subscribers can be increased. This is due to the behavior of DASA under overload conditions.

Under normal conditions, where there are sufficient resources exist to satisfy demands in a timely manner, all of the calls are completed and activities are scheduled essentially in order of their respective deadlines regardless of their relative priorities. When demand exceeds the supply of shared resources (even within a single switch), some calls cannot be completed. In that case, a call's priority would be considered when making scheduling decisions so that more important calls receive shared resources at the expense of less important calls. In fact, DASA would abort less important calls that are holding shared resources in order to free circuits and switches to complete new, higher priority calls⁴³.

⁴³While aborting any calls is unfortunate — they represent a disruption of service to customers — these aborts will not entail serious damage. More likely, an abort will result in a disgruntled caller and callee. And since humans are involved in virtually every call, they are capable of taking appropriate steps following an aborted call — perhaps redialing immediately, maybe waiting awhile before redialing, or maybe just waiting for a more opportune time if the call was not at all urgent. The actual abort processing presents the telephone company with an opportunity to make the abort in a fairly painless way. As the connection is broken, each party in the call could be informed that the call had to be aborted in favor of an urgent call. Furthermore, the effected parties could be given some compensation for their inconvenience such as an account credit or a free call at a later date. The processing requirements for this type of abort processing could be associated with the acquisition of circuits and switch connections and is accommodated by the abort model for this research, which allows a resource-dependent amount of processing time to be reserved in case an abort occurs.

The transition from overload to normal (non-overload) processing would be as graceful as the transformation into the overload case, where most parties are likely to be unaffected while emergency traffic acquires the resources it requires.

Without presenting them here, there are a wealth of statistics available to the telephone companies describing the frequency at which calls arrive throughout a day, profiling various days (weekends, weekdays, Mother's Day, and so on), the computational requirements to route a call and to make the necessary connections, and the numbers and types of the shared resources in the system (circuits, switch connections, logs, and databases for instance). These statistics would be used to consult the simulation charts presented in this chapter or others derived for this specific application.

5.3.1.2. Process Control: A Steel Mill

A steel mill provides a number of supervisory control applications that could benefit from advantageous real-time scheduling. This section returns to the example that was originally introduced in Sections 1.1.2 and 1.4, focusing on a computer system that controls a number of furnaces supplying steel with specific compositions to a pair of continuous casters, which cast the molten steel into slabs.

First, consider the time constraints for this application.

Each caster continuously produces a slab that is cut to specified lengths to fill orders. The slab lengths typically vary between twenty and forty feet, and each time a new slab is cut, a new slab record has to be generated and stored.

The caster speed varies — if it moves too quickly, the metal will not have solidified sufficiently by the time it emerges from the caster; if it moves too slowly, productivity will be unnecessarily low. The caster is operated at the maximum speed at which solid steel can be produced. This speed is determined by the temperature and chemistry of the steel being cast, the water temperature and spray rates of cooling nozzles located along the length of the caster, and several other factors. Typically, a new foot of steel emerges from the caster every six to twelve seconds.

Each time a new foot of steel is cast, a record must be created to document the chemistry of the foot and other information that is used to track the metal through the mill. If this information is lost or is not recorded on time, the chemistry of the slab cannot be adequately certified for customers with strict product quality requirements, and the slab cannot be sold to them. The processing that occurs as each foot of steel is cast is quite complex and requires a second or two of processing time.

The furnace has fewer tight time constraints than the caster. The furnaces produce steel in units called "heats." A heat typically requires between thirty and forty-five minutes to produce. During that time, the chemistry of the steel is calculated several times by a complex analytical model. The chemistry is also measured directly by a chemistry laboratory. Even after the heat is produced the steel's composition may be adjusted at a liquid metallurgy facility. Near the conclusion of a heat, oxygen is blown through the molten metal to reduce the carbon content of the steel. It is important to produce steel with a fairly precise

carbon content because of the extent to which carbon content affects the physical properties of steel. The oxygen is blown through the steel under the direction of the supervisory control computer, and it must be shut off at a precise time after it has started. This time is determined by the supervisory control computer based on the analytic model and the measured chemical composition of the steel. Missing this deadline can be costly.

Next, consider the shared resources in this example.

Each heat is tracked as it makes its way through the mill from the furnace to the caster and beyond. The primary database for this tracking is called the heat log. An entry in the heat log is initially made as the furnace begins a heat. The record may be modified by the liquid metallurgy facility or a holding station or even one of the casters. Information arrives for the heat log asynchronously. There is typically, for example, no guaranteed response time for the chemistry laboratory to return an analysis; heats are not produced periodically, although they are produced regularly; and the order in which heats are cast can change on very short notice.

There are a number of other databases in this example. All are shared among multiple activities. Usually, most of these activities are cooperating to produce steel, while others perform maintenance tasks, such as calculating the lifetime of furnace linings and cutting torches or monitoring the inventory of scrap metal and critical ingredients. All of these activities require access to the databases.

Often activities cooperate to carry out the various application tasks. These tasks, perhaps fifty or sixty in number, make extensive use of signals to communicate with one another. Typically, a number of activities cannot proceed until one or more other activities have properly gathered and prepared the necessary data or until some external event has occurred. Signals are an efficient communication mechanism in such systems.

Devices are also shared in this application. The communication channels to the lower-level process control computers, to the higher-level production control computers, and to human operators that oversee production are of particular interest.

Notice that this application fits the model outlined in this thesis. The mill exists to make steel, which has a very definite value. It is possible to place corresponding values on the steps taken to produce the steel, making the use of time-value functions feasible for this application.

Furthermore, it is a supervisory control application with deadlines that are on the order of seconds. All of the component physical processes proceed asynchronously, and the processor utilization is sufficiently high that some transient overloads will occur.

Of course, failures in the system or alarm conditions from the lower-level process control computers can also add unanticipated load to the supervisory control system for a generally unspecified length of time. In addition, queries and commands from human operators also contribute to the processing load. They arrive asynchronously and typically must be serviced within a matter of seconds. Overloads are not unusual in these systems.

Chapter 6

Related Work and Current Practice

There has been a great deal of research done on scheduling, in general, and scheduling for real-time systems, in particular, through the years. This chapter will attempt to put this thesis work in its proper place within this overall context.

A wide variety of scheduling algorithms have been devised and analyzed through the years for computers. Most of the basic scheduling algorithms are covered in text books on scheduling [Baker 74, French 82] or on operating systems [Janson 85, Peterson 85]. Each algorithm possesses certain properties that differentiate it from others. For instance, round-robin is fair, while shortest processing time first maximizes throughput. However, many of these properties have no value in real-time systems. Nonetheless, these texts do contain scheduling algorithms that are useful in real-time systems.

Real-time scheduling algorithms can be categorized in a number of ways. For now, the algorithms will be divided into two groups: those that are priority-based and those that are deadline-based.

6.1. Priority-Based Scheduling

Most of the real-time systems currently in service employ a static priority scheduler of one type or another. In these systems, component activities are assigned static priorities, and the systems are tuned so that they will typically meet their time constraints. There is also a large body of literature that has investigated priority-based scheduling algorithms beyond this current practice. In [Liu 73], a method for static priority assignment was presented for periodic real-time activities. The scheduling discipline that has grown from this work is called *rate monotonic scheduling*. This basic approach has been elaborated and expanded upon since (e.g., [Sha 86]), but the applications for which it is intended are always those where most, if not all, of the activities are periodic, and where the periodic activities are always the most important activities in the system. While there are systems that fit this description, the family of supervisory control systems that are of interest in this thesis do not. Also, none of the rate monotonic scheduling algorithms deal directly with the problem of scheduling a set of dependent activities.

A second class of priority-based scheduling algorithms has dealt explicitly with some of the scheduling difficulties that arise as a result of the dynamic interaction of activities. Some operating systems (e.g., VMS [KB 84]) implement priority adjustment schemes to refine the simple static priority model, and other schemes have been proposed in the literature as well ([Sha 87]). All of these schemes address problems in

which a lower priority activity that shares a resource with a higher priority activity can block the higher priority activity for an arbitrarily long time. The solution, roughly speaking, allows the lower priority activity to assume a higher priority for at least long enough to complete its access to the shared resource, thereby allowing the higher priority activity to resume. This approach does solve some problems that are associated with simple priority-based scheduling algorithms, but it does not come to grips with the fundamental shortcoming of all of the priority-based schemes: priorities are unable to adequately capture the critical scheduling information for activities. Specifically, an individual activity's importance to the overall application *and* its urgency are two independent factors: an activity is not urgent just because it is very important, and it is not important just because it is urgent. This distinction is lost in static priority scheduling schemes where both importance and urgency must be reflected in a single quantity, the activity's priority.

6.2. Deadline-Based Scheduling

The second group of real-time scheduling algorithms to be dealt with are the deadline-based algorithms⁴⁴. These algorithms seem well-suited for real-time systems since they explicitly take into account an activity's time constraints, and they do not typically require that all activities be periodic. Deadline schedulers have been in use in operating systems at least since the 1960s, and [Liu 73] demonstrated the optimality of deadline scheduling under one computational model. Unfortunately, the basic deadline scheduling algorithm becomes unstable whenever an overload occurs; it acts to minimize the maximum job lateness and maximum job tardiness ([Conway 67]). This may be the desired action, but often it is not. Consequently, a great deal of work has been done to modify the behavior of deadline scheduling in overload situations. Some work that does not consider dependency requirements includes: [Martel 82], which presents an algorithm that will complete all of the (independent) activities while minimizing the maximum lateness of any individual activity; [Moore 68], which uses a scheme that also completes all of the independent activities while minimizing the maximum deferral cost associated with any activity; and [Locke 86], which does not necessarily execute all of the activities, but does attempt to maximize the value acquired by completing those that are executed. In each case, these schemes do not consider dependencies, but do address the issue of overload handling, which is one of the main interests of this thesis.

Historically, there has been a great deal of emphasis placed on being able to guarantee that deadlines can be met. In simple systems that have been built, this has been possible, or has, at least, appeared to be possible. As systems have grown, this has become increasingly more difficult to do. In large, dynamic systems, it is rapidly becoming impossible. Nonetheless, guaranteeing that deadlines can be met is often considered a prime requirement for so called *hard* real-time systems, and much work has been done in this area. (In a hard real-time system, missing even a single deadline means that the entire system has failed.) For simple systems where all of the activities need to be scheduled periodically and have fixed execution time requirements, [Liu 73] and others allow an off-line analysis to guarantee the schedulability of a set of activities under certain assumptions. In more dynamic cases where less emphasis is placed on periodic

⁴⁴In some of the management and operations research literature, deadlines are referred to as due dates.

activities, work similar to [Ramamritham 84] attempts to provide the same type of guarantee. However, it is not obvious that attempting to offer true guarantees is wise in a dynamic system because honoring a guarantee may result in an inability to schedule a new activity that is clearly more important and more urgent than the previously guaranteed activity. In addition, the guarantees that are offered are not absolute. Receiving a guarantee indicates that adequate resources have been reserved to complete an activity by the desired time. If resources are subsequently lost — due to a processor or a power failure, for example — the guarantees made cannot always be met.

There is also a body of literature that explicitly deals with dependencies in deadline-based scheduling algorithms. It should be noted that what is termed a dependency consideration in this thesis encompasses both the notion of a precedence constraint (e.g., activity A_1 must complete before either activity A_2 may begin) and the notion of a resource requirement (e.g., activity A_1 requires exclusive use of resource R for time T during its execution). In the literature, these two types of dependencies are often treated separately. [Blazewicz 77], for instance, deals only with precedence constraints and provides an algorithm that will allow activities with different arrival times and known, fixed precedence constraints to be scheduled in a hard real-time system. This algorithm can be thought of as a deadline inheritance algorithm, whereby an activity is scheduled as if it had a deadline "close" to that possessed by another activity that both depends on it and has a nearer deadline.⁴⁵ Unfortunately, these precedence constraints are fixed, making a straightforward extension to handle resource requirements difficult. Also, no effort is made to handle overload cases, since, by Blazewicz's definition, a missed deadline means that the entire system has failed.

[Cheng 86] looks only at precedence constraints, while [Zhao 87] looks at both precedence constraints and resource requirements. In both cases, these represent extensions of [Ramamritham 84] and share the same shortcomings — they attempt to make guarantees to run specific activities at the possible expense of more urgent or more important activities that may arrive later, and the guarantees are not truly guarantees since unanticipated problems can prevent their fulfillment. In addition, although [Zhao 87] presents a more dynamic, less restrictive model than that presented in most of the work in this area, knowledge of the specific resource requirements of any activity to be run is still assumed to be known in advance.

[Lawler 73] deals with precedence constraints when scheduling a group of activities on a single machine and presents an algorithm that uses a monotone cost function to derive a schedule that minimizes the maximum of the incurred costs. However, the activities to be scheduled have no deadlines, nor do they have any resource requirements. [Elsayed 82] presents heuristics to schedule a set of activities that share resources to complete a project. Once again, there are no deadlines associated with any of the activities.

Some of the previous references deal with uniprocessor scheduling, and some deal with multiprocessor or multiple processor scheduling. This distinction was not made previously because the number of

⁴⁵In fact, one view of the DASA algorithm to be examined in the thesis work is exactly this. It incorporates the idea that the activities on which some activity depends must be dealt with before the activity's deadline, as must the activity itself. However, in addition, the algorithm assesses the situation to decide if there is currently an overload, and if so, selects the subset of activities to be run according to a meaningful metric.

processors, although certainly an important consideration⁴⁶, is of secondary concern for the work at hand. The primary issues being addressed when comparing and contrasting those efforts with this one are: whether or not time constraints are dealt with explicitly, the amount and type of information on which scheduling decisions are based, and the fundamental nature of applications (whether they are static or dynamic, periodic or aperiodic; whether overloads can occur and if so how they are handled). And, although a great deal of work has touched on various aspects of the thesis problem, none of this work has addressed all of the key issues at once.

6.3. Other Related Work

The computational model presented in this thesis provides for the abortion of an activity. This is done for two reasons. First of all, in any application, if an activity that manipulates shared resources is to be terminated, unless specific steps are taken there is a danger that the shared resources will either be unavailable for use by other activities or left in an inconsistent state. The abort mechanism addresses this problem by allowing the shared resources to be returned to an acceptable state for later use. Secondly, an abort mechanism similar to that just mentioned can be used to support an atomic transaction facility [Eswaran 76]. The ability to include such a powerful facility in real-time systems is inviting⁴⁷, and the work presented here can assist in making this feasible at some point. Some work has already been done along those lines. Often, this has involved changing the concurrency control features found in traditional database transaction managers ([Liskov 83, McKendry 85, Sha 85]). Other work has examined the problem of scheduling transactions using the standard concurrency control rules. However, the models chosen for work in this area ([Liu 88], for example) usually require detailed prior knowledge of the precise resource requirements and exact access and release timings for each resource in each transaction. This thesis addresses a more dynamic model than that.

Finally, a few other research directions should be mentioned to put this work in its proper context. An underlying assumption of this work is that dependencies among component activities are a natural product of complex, dynamic real-time systems. There is some work that attempts to approach the construction of applications from other points of view. [Herlihy 88] explores an approach that would eliminate the need for any activity to wait on other activities when accessing resources. However, this approach does not allow the maintenance of mutually consistent resources, which is often important in real-time systems. [Birman 88, Joseph 88] outline portions of a scheme that allows application-specific consistency constraints to be satisfied by utilizing a set of communication and replication mechanisms. How to specify the behavior of objects that have been composed in this way so that large applications can be constructed using a modular design methodology is an important open question with respect to this approach.

⁴⁶Note, for instance, that a scheduling algorithm that is optimal for a uniprocessor may not be optimal for a multiprocessor. A simple deadline scheduler with no overloads demonstrates this fact.

⁴⁷In fact, [Jensen 76] suggests using transactions not only for real-time applications, but also within a decentralized operating system that supports these applications.

- [Bach 86] Bach, M. J.
The Design of the UNIX Operating System.
Prentice-Hall, Inc., 1986.
- [Baker 74] Baker, K. R.
Introduction to Sequencing and Scheduling.
John Wiley & Sons, 1974.
- [Bennett 88] Bennett, S.
Prentice Hall International Series in Systems and Control Engineering: Real-Time Computer Control: An Introduction.
Prentice Hall, 1988.
- [Birman 88] Birman, K. P. and Joseph, T. A.
Exploiting Replication.
Technical Report TR 88-917, Cornell University, Department of Computer Science, Ithaca, NY, June, 1988.
This is a preprint of material that will appear in the collected lecture notes from 'Arctic 88, An Advanced Course on Operating Systems', Tromso, Norway, July 5-14, 1988.
The lecture notes will appear in book form later this year.
- [Blazewicz 77] Blazewicz, J.
Scheduling Dependent Tasks with Different Arrival Times to Meet Deadlines.
Modelling and Performance Evaluation of Computer Systems.
North-Holland Publishing Company, 1977.
Proceedings of the International Workshop organized by the Commission of the European Communities, Joint Research Centre, Ispra Establishment, Department A, Ispra (Varese), Italy, October 4-6, 1976.
- [Cheng 86] Cheng, S., Stankovic, J. A. and Ramamritham, K.
Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems.
In *Proceedings of the Real-Time Systems Symposium*, pages 166-174. December, 1986.
- [Conway 67] Conway, R. W., Maxwell, W. L. and Miller, L. W.
Theory of Scheduling.
Addison-Wesley Publishing Company, 1967.
- [Elsayed 82] Elsayed, E. A.
Algorithms for Project Scheduling with Resource Constraints.
International Journal of Production Research 20(1):95-103, January/February, 1982.
- [Eswaran 76] Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-633, November, 1976.
- [French 82] French, S.
Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop.
John Wiley & Sons, 1982.
- [GD 80] General Dynamics.
Computer Program Product Specification for the System Function Processor Operational Flight Program for the F-16 Multinational Staged Improvement Program, Block 30.
Technical Report CPCI 7175-1A00, General Dynamics Corporation, December, 1980.
- [Herlihy 88] Herlihy, M. P.
Impossibility and Universality Results for Wait-Free Synchronization.
Technical Report CMU-CS-88-140, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, May, 1988.

- [Janson 85] Janson, P. A.
Operating Systems: Structures and Mechanisms.
Academic Press, 1985.
- [Jensen 75] Jensen, E. D.
Time-Value Functions for BMD Radar Scheduling.
Technical Report, Honeywell Systems and Research Center, June, 1975.
- [Jensen 76] Jensen, E. D.
Decentralized Operating Systems.
In *Workshop on Distributed Processing*. Brown University, August, 1976.
- [Joseph 88] Joseph, T. A. and Birman, K. P.
Reliable Broadcast Protocols.
Technical Report TR 88-918, Cornell University, Department of Computer Science,
Ithaca, NY, June, 1988.
This is a preprint of material that will appear in the collected lecture notes from 'Arctic
88, An Advanced Course on Operating Systems', Tromso, Norway, July 5-14, 1988.
The lecture notes will appear in book form later this year.
- [KB 84] Kenah, L. J. and Bate, S. F.
VAX/VMS Internals and Data Structures.
Digital Press, 1984.
- [Lawler 73] Lawler, E. L.
Optimal Sequencing of a Single Machine Subject to Precedence Constraints.
Management Science 19(5):544-546, January, 1973.
- [Liskov 83] Liskov, B. and Scheifler, R.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.
- [Liu 73] Liu, C. L. and Layland, J. W.
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.
Journal of the Association for Computing Machinery 20(1):46-61, January, 1973.
- [Liu 88] Liu, J. W. S., Lin, K. J. and Song, X.
Scheduling Hard Real-Time Transactions.
The Fifth Workshop on Real-Time Software and Operating Systems :112-116, May,
1988.
- [Locke 86] Locke, C. D.
Best-Effort Decision Making for Real-Time Scheduling.
PhD thesis, Carnegie Mellon University, May, 1986.
- [Mach 86] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young,
M.
Mach: A New Kernel Foundation for UNIX Development.
In *Proceedings of Summer Usenix*. July, 1986.
- [MacLaren 80] MacLaren, L.
Evolving Toward Ada in Real-Time Systems.
ACM SIGPLAN Notices 15(11):146-155, November, 1980.
This issue was also the Proceedings of the ACM-SIGPLAN Symposium on the Ada
Programming Language, Boston, MA, December 9-11, 1980.
- [Martel 82] Martel, C.
Preemptive Scheduling with Release Times, Deadlines, and Due Dates.
Journal of the Association for Computing Machinery 29(3):812-829, July, 1982.
- [McKendry 85] McKendry, M. S.
Ordering Actions for Visibility.
Transactions on Software Engineering (IEEE) 11(6):509-519, June, 1985.

- [Moore 68] Moore, J. M.
An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs.
Management Science 15(1):102-109, September, 1968.
- [Northcutt 87] Northcutt, J. D.
Perspectives in Computing. Volume 16: *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*.
Academic Press, 1987.
- [Peterson 85] Peterson, J. L. and Silberschatz, A.
Operating System Concepts, Second Edition.
Addison-Wesley Publishing Company, 1985.
- [Ramamritham 84] Ramamritham, K. and Stankovic, J. A.
Dynamic Task Scheduling in Hard Real-Time Distributed Systems.
IEEE Software 1(3):65-75, July, 1984.
- [Rauch-Hindin 87] Rauch-Hindin, W. B.
UNIX Overcomes Its Real-Time Limitations.
UNIX World 4(11):64-78, November, 1987.
- [Ritchie 74] Ritchie, D. M. and Thompson, K.
The UNIX Time-Sharing System.
Communications of the ACM 17(7):365-375, July, 1974.
- [Sha 85] Sha, L.
Modular Concurrency Control and Failure Recovery --- Consistency, Correctness and Optimality.
PhD thesis, Carnegie Mellon University, 1985.
- [Sha 86] Sha, L., Lehoczky, J. P. and Rajkumar, R.
Solutions for Some Practical Problems in Prioritized Preemptive Scheduling.
In *Proceedings of the Real-Time Systems Symposium*, pages 181-191. December, 1986.
- [Sha 87] Sha, L., Rajkumar, R. and Lehoczky, J. P.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization.
Technical Report CMU-CS-87-181, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, December, 1987.
- [Stadick 83] Stadick, E. M.
A Real-Time Control System Implementation Study Using the Ada Programming Language.
Technical Report NSWC TR-83-213, Naval Surface Weapons Center, 1983.
- [Ullman 75] Ullman, J. D.
NP-Complete Scheduling Problems.
Journal of Computer and System Sciences 10(3):384-393, June, 1975.
- [Zhao 87] Zhao, W., Ramamritham, K., and Stankovic, J. A.
Scheduling Tasks with Resource Requirements in Hard Real-Time Systems.
IEEE Transactions on Software Engineering SE-13(5):564-577, May, 1987.

Appendix A

The General Scheduling Automaton Framework

In order to provide a formal framework in which to discuss scheduling policies for real-time activities, the following model has been adopted.

Notation The following conventions, modeled after the style used by Maurice Herlihy, are employed in defining the computational model and the automaton that will examine schedules:

- Identifiers written in all capital letters denote domains of values (e.g., `TIMESTAMP`)
- The automaton that evaluates schedules has certain state components associated with it; these are designated by identifiers that begin with a single capital letter followed immediately by one or more lower-case letters (e.g., `Total`, `AbortClock`)
- Operations are accepted by the automaton if they meet certain preconditions
- Following operation execution, certain postconditions hold; when these result in modifying the value of a state component, the new value is followed by an apostrophe (e.g., `Clock' = Clock + 1`)

The Model

1. An application is composed of a set of activities, each of which comprises a sequence of computational phases. At any given time, these activities can be referred to by means of the phase that they are currently carrying out. Therefore the set of activities can be represented by the set of phases currently defined: $\{p_0, p_1, p_2, \dots\}$
2. While executing an application, an observer located within the operating system could monitor a sequence of time-stamped events passing to and from the scheduler. These events are of the form:

$$t_{event} \ op(parms) \ O$$

where,

t is a timestamp,
 op is the operation associated with the event (as defined below),
 $parms$ are the arguments for the operation,
 O is the originator of the event (either p , for a phase, or S , for the scheduler)

A sequence of these events is called a history. Notice that some of these events are generated by individual phases and some are generated by the scheduler.

3. make a real table] The operations that may occur in events, and the potential originators of each, include:

Event	Potential Originator(s)
• <i>request-phase</i> (<i>v</i> , <i>t_{expected}</i>)	Phase
• <i>abort-phase</i> (<i>p</i>)	Scheduler or Phase
• <i>preempt-phase</i> (<i>p</i>)	Scheduler
• <i>resume-phase</i> (<i>p</i>)	Scheduler
• <i>request</i> (<i>r</i>)	Phase
• <i>grant</i> (<i>p</i> , <i>r</i> , <i>t_{undo}</i>)	Scheduler

4. The individual computational phases that comprise an activity are delimited by 'request-phase' events. A 'request-phase' event ends one computational phase of an activity and begins the next atomically.
5. Phases may access shared resources. A request for such access is signalled by a phase by means of a 'request' event for the specific resource desired. Permission to access a shared resource is signalled to the phase by means of a 'grant' event.
6. All shared resources that are held by an activity must be released at the completion or abortion of each computational phase.
7. At any given time there is one phase that is active. It may be preempted by the scheduler. This is signalled by a 'preempt-phase' event. The scheduler may subsequently determine that the phase should be resumed; this is signalled by a 'resume-phase' event.
8. A *history* is defined as a sequence of events. Not all histories are meaningful or well-formed. Let $e_0, e_1, e_2, \dots$ denote events. Then, formally, a history, H , can be denoted as:

$$H = e_0 \cdot e_1 \cdot e_2 \cdot \dots \cdot e_n$$

where the operator "." denotes concatenation.

Informally, a projection of a history selects certain events from a history, preserving their relative positions in the projection. Therefore, a projection of a history could include all of the 'request-phase's from the history or all of the events that dealt with a specific phase. The symbol "|" denotes a projection. So for example, $H|p$ represents the projection of history H onto phase p . This projection would include all of the events that were originated by phase p or that were originated by the scheduler and included p as an operational parameter.

9. Some additional terminology and notation will be useful for discussing events. Let an event, e represent the following event:

$$e = t_{event} \quad op(parms) \quad O$$

Then define the following simple functions:

$$timestamp(e) = t_{event}$$

$$eventtype(e) = op$$

$$source(e) = O$$

10. The conditions that define a well-formed history include⁴⁸:

- event timestamps must increase monotonically and must be unique — *test*: examine the timestamps on events; for example, apply the function `timestampsOK()` to a history H to verify that it meets this requirement, where `timestampsOK()` is defined as:

$$\text{timestampsOK}(\emptyset) = \text{timestampsOK}(e) = \text{true}$$

$$\text{timestampsOK}(e_1 \cdot e_2 \cdot H) = \begin{array}{ll} \text{false,} & \text{if } \text{timestamp}(e_1) \\ & \geq \text{timestamp}(e_2) \\ \text{timestampsOK}(H), & \text{otherwise} \end{array}$$

- request for a resource must appear in the schedule before the corresponding grant — *test*: for each 'grant' event, search the history of the phase in which the 'grant' occurred for a preceding 'request' for the same resource
- a phase cannot be preempted if it is not active; it cannot be resumed if it is active; and so on — *simple tests check all of these conditions*
- a given phase either commits or aborts; the events assure that a single phase cannot do both; however, a well-formed history must have at most one 'abort-phase' event for any given phase — *test*: examine the history for the occurrence of two or more 'abort-phase' events for a single activity that are not separated by a 'request-phase' event.
- expected compute time is accurate — *test*: check that the estimated computation time equals the actual computational time used; for example, the following test could be applied:

$$\text{ctest}(H) = (\forall p)(\text{comptimeOK}(H \mid p) \vee \text{phaseaborted}(H \mid p) \vee \text{phaseunfinished}(H \mid p))$$

where,

⁴⁸It is not always clear that a specific test be a requirement of a well-formed history or whether it is a requirement that determines which histories will be accepted by a given automaton. There is no question that the proper temporal ordering of events is a requirement for a well-formed history; however, tests that constrain the relative ordering of specific events — for instance, 'request' and 'grant' events — in a history are not so obviously requirements for a well-formed history. As a result, this list is merely an attempt to lay down an initial set of tests. Some of these tests need not be done prior to submitting the history to an automaton — in those cases, the automaton will enforce the requirements verified by the tests in question.

$$\text{comptimeOK}(p.o) = \text{comptimeOK}(p.e) = 0$$

$$\begin{aligned} \text{comptimeOK}(p.e_1.e_2.H) = & t_2 - t_1 + \text{comptimeOK}(H), \\ & t_2 - t_1, \end{aligned} \quad \begin{aligned} & \text{if } (e_1 = t_1 \text{ resume-phase}(p) S \\ & \quad \vee e_1 = t_1 \text{ grant}(p) S) \\ & \wedge (e_2 = t_2 \text{ preempt-phase}(p) S \\ & \quad \vee e_2 = t_2 \text{ request}(r) p) \\ & \text{if } (e_1 = t_1 \text{ resume-phase}(p) S \\ & \quad \vee e_1 = t_1 \text{ grant}(p) S) \\ & \wedge (e_2 = t_2 \text{ request-phase}(v.t) p \\ & \quad \vee e_2 = t_2 \text{ abort-phase}(p) O) \end{aligned}$$

$$\text{phaseaborted}(p.o) = \text{false}$$

$$\begin{aligned} \text{phaseaborted}(p.e.H) = & \text{true,} \\ & \text{false,} \\ & \text{phaseaborted}(p.H), \end{aligned} \quad \begin{aligned} & \text{if } e = t \text{ abort-phase}(p) O \\ & \text{if } e = t_1 \text{ request-phase}(v.t_2) p \\ & \text{otherwise} \end{aligned}$$

$$\text{phaseunfinished}(p.o) = \text{true}$$

$$\begin{aligned} \text{phaseunfinished}(p.e.H) = & \text{false,} \\ & \text{phaseunfinished}(p.H), \end{aligned} \quad \begin{aligned} & \text{if } e = t \text{ abort-phase}(p) O \\ & \vee e = t_1 \text{ request-phase}(v.t_2) p \\ & \text{otherwise} \end{aligned}$$

- expected abort time is accurate — *test: similar to the previous test*
- estimated computation time required for a phase must always be greater than or equal to zero⁴⁹ — *test: straightforward inspection of each 'request-phase' event in the history*
- no 'request' event should request shared access to the nullresource — *test: straightforward inspection of each 'request' event in the history*

11. The state components associated with the scheduling automaton framework are:

- ExecMode: PHASE $\rightarrow$ MODE (MODE is either 'normal' or 'abort')
- ExecClock: PHASE $\rightarrow$ VIRTUAL-TIME
- AbortClock: PHASE $\rightarrow$ VIRTUAL-TIME
- ResumeTime: PHASE $\rightarrow$ TIMESTAMP
- Value: PHASE $\rightarrow$ (TIMESTAMP $\rightarrow$ VALUE)
- Total: VALUE (initially '0')
- RunningPhase: PHASE (initially 'nullphase')
- PhaseElect: PHASE (initially '<normal, nullphase>')
- PhaseList: list of PHASE (initially 'o')
- Other state components are also associated with the automaton. These are used to

⁴⁹An additional requirement may also be placed on the parameters of a 'request-phase' event: the value function must be of the appropriate form, as outlined below. This requirement has not been included in this list because the tests that are present all apply to the general case of scheduling with dependency considerations in a real-time environment using information available from arbitrary time-value functions. This requirement is related to a simplification made to make the work more clear and more manageable, and so does not seem to carry the same weight as the others listed above.

handle some of the bookkeeping details for the specific scheduler being used. The components that appear above are intended to reflect the state that any specific scheduler would need and maintain under this general model.

Specific initial values may be given to many of these state components in order to satisfy the requirements of a given automaton.

12. Operations recognized by the automaton and their general minimal/skeletal preconditions and postconditions:

• t_{event} request-phase($v, t_{expected}$) p :

preconditions:

true <No preconditions here so that interrupts and other new phases can occur at any time>

postconditions:

if (RunningPhase = p) then
 if (ExecMode(p) = normal) then
 Total' = Total + Value(p)(t_{event})
 else
 ;no value for aborted phase
 ;release the resources acquired during the phase

;accept values for scheduling parameters

Value'(p) = v

ExecClock'(p) = $t_{expected}$

AbortClock'(p) = 0

ExecMode'(p) = normal

;note that p is not resource-waiting

;make sure p is part of the list of phases, if necessary

if ($t_{expected} > 0$) then

 PhaseList' = PhaseList $\cup$ { p }

else

 PhaseList' = PhaseList - { p }

• t_{event} abort-phase(p) O :

preconditions:

<Specific to the scheduler under consideration>

postconditions:

ExecMode'(p) = abort

ResumeTime'(p) = t_{event}

• t_{event} preempt-phase(p) S :

preconditions:

<Specific to the scheduler under consideration>

postconditions:

if (ExecMode(p) = normal) then

 ExecClock'(p) = ExecClock(p) - (t_{event} - ResumeTime(p))

else

 AbortClock'(p) = AbortClock(p) - (t_{event} - ResumeTime(p))

• t_{event} resume-phase(p) S :

preconditions:

<Specific to the scheduler under consideration>

postconditions:

ResumeTime'(p) = t_{event}

• t_{event} request(r) p :

preconditions:

<Specific to the scheduler under consideration>

postconditions:

$$\text{ExecClock}'(p) = \text{ExecClock}(p) - (t_{\text{event}} - \text{ResumeTime}(p))$$

- $t_{\text{event}} \text{ grant}(p, r, \text{undotime}(r)) S:$

preconditions:

<Specific to the scheduler under consideration>

postconditions:

$$\text{ResumeTime}'(p) = t_{\text{event}}$$

$$\text{AbortClock}'(p) = \text{AbortClock}(p) + \text{undotime}(r)^{50}$$

Specific/Simplifying Assumptions/Restrictions Time-value functions are all of the form:

$$v(t) = (\text{step}(\text{val}, t_c))(t),$$

where,

t_c is the critical time, or deadline, for this phase of an activity,

val is the value associated with completing a phase at any time before its deadline,

$$\text{step}(\text{val}, t_c)(t) = \begin{array}{ll} \text{val}, & t \leq t_c \\ 0, & t > t_c \end{array}$$

⁵⁰The function 'undotime()' indicates the amount of time that will be required to restore the resource just acquired to its current state. This function may vary from system to system and from application to application. Consequently, for the purposes of this work, its place and role have been indicated without applying a single definition for this function.

Appendix B

Derivation of DASA/ND Scheduling Automaton

General/Introduction. When there are no dependency considerations — as when comparing the DASA algorithm to LBEA — some simplifications can be made to the formulae that define DASA. These simplifications aid in allowing direct comparisons to be made between algorithms. The following derivation points out and justifies these simplifications.

In each simplification that follows, the original formula to be simplified is taken directly from the description of the DASA Algorithm. The derivation of the simplification is then offered.

The Functional Definition of *SelectPhase()*.

1. By definition, the fact that there are no dependencies means that there is no interaction or cooperation among phases through shared resources. (Otherwise, there would be a risk of a dependency arising.) In the model presented here, this situation is represented by:

$$(\forall p) \text{ResourceRequested}(p) = \text{nullresource}$$

2. Simplification (1) allows the definition of *Dep()* to be transformed from

$$\text{Dep}(p) = \begin{array}{ll} \text{nullphase}, & \text{if ResourceRequested}(p) \\ & = \text{nullresource}, \\ \text{Owner(ResourceRequested}(p)), & \text{otherwise} \end{array}$$

to

$$\text{Dep}(p) = \text{nullphase}$$

3. Simplification (2) leads directly to the transformation of the definition of the function *dependencylist()* from

$$\begin{aligned} \text{dependencylist}(p) = & \begin{array}{ll} \emptyset, & \text{if } \text{Dep}(p) = \text{nullphase} \\ \text{dependencylist}(\text{Dep}(p)) \cup \{ \langle \text{normal}, \text{Dep}(p) \rangle \}, & \text{if } \text{AbortClock}(\text{Dep}(p)) \\ & \geq \text{ExecClock}(\text{Dep}(p)) \\ \{ \langle \text{abort}, \text{Dep}(p) \rangle \}, & \text{otherwise} \end{array} \end{aligned}$$

to

$$\text{dependencylist}(p) = \emptyset$$

4. Simplification (2) also leads to the transformation of the function *PVD()* from

$$\text{PVD}(p) = \begin{array}{ll} 0, & \text{if } \text{ExecMode}(p) = \text{abort}, \\ \frac{\text{Val}(p) + \text{PV}(\text{Dep}(p))}{\text{ExecClock}(p) + \text{PT}(\text{Dep}(p))}, & \text{otherwise} \end{array}$$

to

$$PVD(p) = \begin{array}{ll} 0, & \text{if } ExecMode(p)=abort, \\ \frac{Val(p)}{ExecClock(p)}, & \text{otherwise} \end{array}$$

since

$$PV(p) = \begin{array}{ll} 0, & \text{if } p=nullphase, \\ 0, & \text{if } AbortClock(p) \\ & < ExecClock(p), \\ Val(p)+PV(Dep(p)), & \text{otherwise} \end{array}$$

$$PT(p) = \begin{array}{ll} 0, & \text{if } p=nullphase, \\ AbortClock(p), & \text{if } AbortClock(p) \\ & < ExecClock(p), \\ ExecClock(p)+PT(Dep(p)), & \text{otherwise} \end{array}$$

5. Applying Simplification (3) transforms

$$tobescheduled(P) = \begin{array}{ll} 0, & \text{if } P=\emptyset \\ \{ \langle normal, p \rangle \} \cup dependencylist(p) \cup tobescheduled(P - \{p\}), & \text{if } p \in P \end{array}$$

to

$$tobescheduled(P) = \begin{array}{ll} 0, & \text{if } P=\emptyset \\ \{ \langle normal, p \rangle \} \cup tobescheduled(P - \{p\}), & \text{if } p \in P \end{array}$$

which is further simplified (by means of an inductive proof on the number of elements in P) to

$$tobescheduled(P) = \begin{array}{ll} 0, & \text{if } P=\emptyset \\ \{ \langle normal, p \rangle \mid p \in P \}, & \text{otherwise} \end{array}$$

and finally to

$$tobescheduled(P) = \{ \langle normal, p \rangle \mid p \in P \}$$

6. Consider the definition of *mustcompleteby*(t):

$$mustcompleteby(t, P) = \begin{array}{ll} 0, & \text{if } t < t_{event} \\ \{ p \mid \langle normal, p \rangle \in tobescheduled(P) \wedge Deadline(p) \leq t \}, & \text{otherwise} \end{array}$$

Substituting the definition of *tobescheduled*(t) that was derived in Simplification (5) yields

$$mustcompleteby(t, P) = \begin{array}{ll} 0, & \text{if } t < t_{event} \\ \{ p \mid \langle normal, p \rangle \in \{ \langle normal, q \rangle \mid q \in P \} \wedge Deadline(p) \leq t \}, & \text{otherwise} \end{array}$$

which is equivalent to ...

$$mustcompleteby(t, P) = \begin{array}{ll} 0, & \text{if } t < t_{event} \\ \{ p \mid p \in P \wedge Deadline(p) \leq t \}, & \text{otherwise} \end{array}$$

7. Again, applying Simplification (3) allows

$$\begin{aligned}
\text{mustfinishby}(t, P) = & \\
& \emptyset, & \text{if } P = \emptyset \vee t < t_{\text{event}} \\
& \vee \text{mustcompleteby}(t, P) = \emptyset \\
& \text{reduce}(t, P, \{ \langle \text{normal}, p \rangle \} \cup \text{dependencylist}(p) \cup \text{mustfinishby}(t, P - \{p\})), \\
& \text{if } p \in \text{mustcompleteby}(t, P)
\end{aligned}$$

to become

$$\begin{aligned}
\text{mustfinishby}(t, P) = & \\
& \emptyset, & \text{if } P = \emptyset \vee t < t_{\text{event}} \\
& \vee \text{mustcompleteby}(t, P) = \emptyset \\
& \text{reduce}(t, P, \{ \langle \text{normal}, p \rangle \} \cup \text{mustfinishby}(t, P - \{p\})), \\
& \text{if } p \in \text{mustcompleteby}(t, P)
\end{aligned}$$

Consider $\text{mustfinishby}()$ for $t \geq t_{\text{event}}$.

Prove: In cases in which there are no dependency considerations and for which $t \geq t_{\text{event}}$, $\text{mustfinishby}()$ never returns a set that includes a phase/mode pair for which the mode is *abort*. That is, prove that

$$(\forall P)(pmp \in \text{mustfinishby}(t, P) \rightarrow \text{Mode}(pmp) \neq \text{abort})$$

Proof. This is proven by induction on i , the number of elements in P , the set of phases for which $\text{mustfinishby}()$ is being evaluated.

Basis. $i = 0$. In this case, $P = \emptyset$. Therefore, $\text{mustfinishby}(t, P) = \emptyset$, and the claim is trivially true.

Inductive Step. Assume that the inductive hypothesis holds for all sets of phases with i or fewer elements. Show that it also holds for all sets of phases with $i+1$ elements.

Let P denote a set of phases with $i+1$ elements. According to the definition of $\text{mustfinishby}()$ given above:

$$\begin{aligned}
\text{mustfinishby}(t, P) = & \\
& \emptyset, & \text{if } P = \emptyset \vee t < t_{\text{event}} \\
& \vee \text{mustcompleteby}(t, P) = \emptyset \\
& \text{reduce}(t, P, \{ \langle \text{normal}, p \rangle \} \cup \text{mustfinishby}(t, P - \{p\})), \\
& \text{if } p \in \text{mustcompleteby}(t, P)
\end{aligned}$$

It is given that $t \geq t_{\text{event}}$, and since $i+1 > 0$, $P \neq \emptyset$. Consequently, which of the two cases in the above definition applies is determined solely by the value of $\text{mustcompleteby}(t, P)$.

If $\text{mustcompleteby}(t, P) = \emptyset$, then $\text{mustfinishby}(t, P) = \emptyset$, too, and once again the inductive hypothesis is trivially true.

Otherwise, $\text{mustcompleteby}(t, P) \neq \emptyset$. In that case, let $p_{mc} \in \text{mustcompleteby}(t, P)$.

As shown in Simplification (6), $\text{mustcompleteby}()$ is defined as:

$$\begin{aligned}
\text{mustcompleteby}(t, P) = & \\
& \emptyset, & \text{if } t < t_{\text{event}} \\
& \{p \mid p \in P \wedge \text{Deadline}(p) \leq t\}, & \text{otherwise}
\end{aligned}$$

Since $p_{mc} \in \text{mustcompleteby}(t, P)$ and $p_{mc} \notin \emptyset$, then

$$p_{mc} \in \{p \mid p \in P \wedge \text{Deadline}(p) \leq t\}$$

Therefore, since all of the elements in this set are members of P . . .

$$p_{mc} \in P$$

This allows the value of $\text{mustfinishby}(t, P)$ to be written as . . .

$$\text{mustfinishby}(t, P) = \text{reduce}(t, P, \{ \langle \text{normal}, p_{mc} \rangle \} \cup \text{mustfinishby}(t, P - \{p_{mc}\}))$$

$\text{Reduce}()$ is defined as:

$$\begin{aligned}
 \text{reduce}(t.P.PMP) = & \\
 & \text{reduce}(t.P.PMP - \{\langle \text{abort}, p \rangle\}), & \text{if } \langle \text{abort}, p \rangle, \langle \text{normal}, p \rangle \in PMP \\
 & & \wedge \langle \text{abort}, p \rangle \notin \text{mustfinishby}(t.P) \\
 & PMP, & \text{otherwise}
 \end{aligned}$$

It is given that P has $i+1$ elements, and it has been proven that p_{mc} is one of them. Consequently, $P - \{p_{mc}\}$ has i elements and the inductive hypothesis asserts that ...

$$pmp \in \text{mustfinishby}(t.P - \{p_{mc}\}) \rightarrow \text{Mode}(pmp) \neq \text{abort}$$

Also, since $\text{Mode}(\langle \text{normal}, p_{mc} \rangle) \neq \text{abort}$, the entire argument passed to the function $\text{reduce}()$ contains no phase/mode pairs for which the mode is *abort*. Therefore, the second case in the definition of $\text{reduce}()$ applies, and $\text{reduce}()$ acts as an identity function for this particular set of arguments ...

$$\begin{aligned}
 \text{reduce}(t.P, \{\langle \text{normal}, p_{mc} \rangle\} \cup \text{mustfinishby}(t.P - \{p_{mc}\})) = \\
 \{\langle \text{normal}, p_{mc} \rangle\} \cup \text{mustfinishby}(t.P - \{p_{mc}\})
 \end{aligned}$$

Inserting this fact into the earlier expression for $\text{mustfinishby}(t.P)$ yields ...

$$\text{mustfinishby}(t.P) = \{\langle \text{normal}, p_{mc} \rangle\} \cup \text{mustfinishby}(t.P - \{p_{mc}\})$$

Assume $pmp \in \text{mustfinishby}(t.P)$. Using the definition for $\text{mustfinishby}(t.P)$ that was just presented ...

$$pmp \in \{\langle \text{normal}, p_{mc} \rangle\} \cup \text{mustfinishby}(t.P - \{p_{mc}\})$$

or equivalently ...

$$pmp \in \{\langle \text{normal}, p_{mc} \rangle\} \vee pmp \in \text{mustfinishby}(t.P - \{p_{mc}\})$$

As was noted earlier, the set of phases $P - \{p_{mc}\}$ has i elements, so the inductive hypothesis holds and asserts ...

$$pmp \in \text{mustfinishby}(t.P - \{p_{mc}\}) \rightarrow \text{Mode}(pmp) \neq \text{abort}$$

Yet ...

$$\begin{aligned}
 pmp \notin \text{mustfinishby}(t.P - \{p_{mc}\}) \\
 \rightarrow pmp \in \{\langle \text{normal}, p_{mc} \rangle\} \\
 \rightarrow pmp = \langle \text{normal}, p_{mc} \rangle \\
 \rightarrow \text{Mode}(pmp) \neq \text{abort}
 \end{aligned}$$

Applying the following identity from formal logic

$$(A \rightarrow B) \wedge (\neg A \rightarrow B) \equiv B$$

to the last two implications leads to the conclusion ...

$$\text{Mode}(pmp) \neq \text{abort}$$

The above result was derived by assuming $pmp \in \text{mustfinishby}(t.P)$. Applying another formal logic identity:

$$\text{Atturnstile } B \equiv A \rightarrow B$$

proves ...

$$pmp \in \text{mustfinishby}(t.P) \rightarrow \text{Mode}(pmp) \neq \text{abort}$$

Therefore, the inductive hypothesis holds for all sets of phases P with $i+1$ members, whether or not $\text{mustcompleteby}(t.P)$ is empty. Q.E.D.

Applying this result to the definition of $\text{mustfinishby}()$, once again noting that $\text{reduce}()$ will always act as an identity function since

$$(\forall P)(pmp \in \text{mustfinishby}(t.P) \rightarrow \text{Mode}(pmp) \neq \text{abort})$$

yields ...

$$\begin{aligned}
\text{mustfinishby}(t,P) = & \\
& 0, & \text{if } P = \emptyset \vee t < t_{\text{event}} \\
& \vee \text{mustcompleteby}(t,P) = 0 \\
& \langle \text{normal}, p \rangle \cup \text{mustfinishby}(t, P - \{p\}), & \text{if } p \in \text{mustcompleteby}(t,P)
\end{aligned}$$

Finally, a simple induction on the size of the set P will yield ...

$$\begin{aligned}
\text{mustfinishby}(t,P) = & \\
& 0, & \text{if } P = \emptyset \vee t < t_{\text{event}} \\
& \vee \text{mustcompleteby}(t,P) = 0 \\
& \langle \text{normal}, p \rangle \mid p \in \text{mustcompleteby}(t,P), & \text{otherwise}
\end{aligned}$$

8. In the formulation of the DASA scheduling algorithm, the function *timerequiredby()* is only evaluated with a result from *mustfinishby()* (ignoring the recursive evaluations that are part of the definition of *timerequiredby()*). As a short inductive proof would indicate, in that case *timerequiredby()* can be simplified since (as shown in Simplification (7)) *mustfinishby()* returns no phase/mode pairs that have an *abort* mode. Therefore, *timerequiredby()* never receives an argument containing a phase/mode pair of the form $\langle \text{abort}, p \rangle$, and it can be simplified from ...

$$\begin{aligned}
\text{timerequiredby}(PMP) = & \\
& 0, & \text{if } PMP = \emptyset \\
& \text{ExecClock}(p) + \text{timerequiredby}(PMP - \{\langle \text{normal}, p \rangle\}), & \text{if } \langle \text{normal}, p \rangle \in PMP \\
& \text{AbortClock}(p) + \text{timerequiredby}(PMP - \{\langle \text{abort}, p \rangle\}), & \text{if } \langle \text{abort}, p \rangle \in PMP
\end{aligned}$$

to ...

$$\begin{aligned}
\text{timerequiredby}(PMP) = & \\
& 0, & \text{if } PMP = \emptyset \\
& \text{ExecClock}(p) + \text{timerequiredby}(PMP - \{\langle \text{normal}, p \rangle\}), & \text{if } \langle \text{normal}, p \rangle \in PMP
\end{aligned}$$

9. *Pickone()* is also only evaluated for an argument that is a result returned by evaluating *mustfinishby()*. Once again, since *mustfinishby()* never returns a set containing an element that is a phase/mode pair with an *abort* mode, *pickone()* can be simplified from ...

$$\begin{aligned}
\text{pickone}(PMP) = & \\
& \langle \text{normal}, p \rangle, & \text{if } \langle \text{normal}, p \rangle \in PMP \\
& & \wedge \text{Dep}(p) = \text{nullphase} \\
& \langle \text{abort}, p \rangle, & \text{if } \langle \text{abort}, p \rangle \in PMP \\
& & \wedge \neg (\exists q) (\langle \text{normal}, q \rangle \in PMP \\
& & \wedge \text{Dep}(q) = \text{nullphase}) \\
& \langle \text{normal}, \text{nullphase} \rangle, & \text{otherwise}
\end{aligned}$$

to ...

$$\begin{aligned}
\text{pickone}(PMP) = & \\
& \langle \text{normal}, p \rangle, & \text{if } \langle \text{normal}, p \rangle \in PMP \\
& & \wedge \text{Dep}(p) = \text{nullphase} \\
& \langle \text{normal}, \text{nullphase} \rangle, & \text{otherwise}
\end{aligned}$$

Since, according to Simplification (2), $(\forall p) \text{Dep}(p) = \text{nullphase}$...

$$\begin{aligned}
\text{pickone}(PMP) = & \\
& \langle \text{normal}, p \rangle, & \text{if } \langle \text{normal}, p \rangle \in PMP \\
& \langle \text{normal}, \text{nullphase} \rangle, & \text{otherwise}
\end{aligned}$$

Finally, this function can be rewritten as ...

$$\begin{aligned} \text{pickone}(PMP) = & \\ & \langle \text{normal}, \text{nullphase} \rangle, & \text{if } PMP = \emptyset \\ & \langle \text{normal}, p \rangle \mid \langle \text{normal}, p \rangle \in PMP, & \text{otherwise} \end{aligned}$$

10. As shown in Simplification (4) above ...

$$\text{PVD}(p) = \begin{aligned} & 0, & \text{if } \text{ExecMode}(p) = \text{abort}, \\ & \frac{\text{Val}(p)}{\text{ExecClock}(p)}, & \text{otherwise} \end{aligned}$$

As shown in Simplification (9), *pickone()* will never return a phase/mode pair as a result whose mode is *abort*. As a result, the precondition for accepting an 'abort-phase' event for the DASA automaton will never be satisfied. Since the postconditions of 'abort-phase' are the only way that 'ExecMode' can be changed to 'abort' for any phase, then ...

$$(\forall p) \text{ExecMode}(p) = \text{normal}$$

This allows the first case in the definition of *PVD()* to be dropped, yielding ...

$$\text{PVD}(p) = \frac{\text{Val}(p)}{\text{ExecClock}(p)}$$

The Simplified Definition of the Automaton. There are also a set of simplifications that can be made to the automaton itself when there are no dependencies to consider. Each of these simplifications are discussed in turn.

1. As pointed out before, all of the simplifications stem from the fact that ...

$$(\forall p) \text{ResourceRequested}(p) = \text{nullresource}$$

Consider the postconditions defined for a 'request' event:

$$\begin{aligned} \text{ExecClock}'(A) &= \text{ExecClock}(A) - (t_a - \text{ResumeTime}(A)) \\ \text{ResourceRequested}'(A) &= r & ; \text{indicate } A \text{ is resource-waiting} \\ \text{PhaseElect}' &= \text{SelectPhase}(\text{PhaseList}) \\ \text{RunningPhase} &= \text{nullphase} & ; \text{give up processor until 'grant'ed resource} \end{aligned}$$

They necessarily include an assignment to *ResourceRequested* for some phase, it must be the case that no 'request' event can be accepted by the simplified DASA automaton. Therefore, the precondition for the acceptance of a 'request' event is *false*, and the event can be eliminated from the automaton.

2. Similarly, consider the precondition for the acceptance of a 'grant' event:

$$\begin{aligned} & (\text{RunningPhase} = \text{nullphase}) \wedge (\text{Phase}(\text{PhaseElect}) = A) \wedge (r \neq \text{nullresource}) \\ & \wedge (\text{ResourceRequested}(\text{Phase}(\text{PhaseElect})) = r) \wedge (\text{Mode}(\text{PhaseElect}) = \text{normal}) \end{aligned}$$

Since it includes as conjuncts $(\text{ResourceRequested}(\text{Phase}(\text{PhaseElect})) = r)$ and $(r \neq \text{nullresource})$, this precondition can never be satisfied because $(\forall p) \text{ResourceRequested}(p) = \text{nullresource}$. Therefore, this precondition will always be *false*, a 'grant' event can never be accepted, and the event can be eliminated from the simplified DASA automaton.

3. Consider the precondition for the acceptance of a 'resume-phase' event:

$$\begin{aligned} & (\text{RunningPhase} = \text{nullphase}) \wedge (\text{Phase}(\text{PhaseElect}) = A) \wedge (\text{Phase}(\text{PhaseElect}) \neq \text{nullphase}) \\ & \wedge \neg \text{ResourceWaiting}(\text{Phase}(\text{PhaseElect})) \wedge (\text{Mode}(\text{PhaseElect}) = \text{normal}) \end{aligned}$$

In particular, consider the conjunct $\neg \text{ResourceWaiting}(\text{Phase}(\text{PhaseElect}))$, remembering that, by definition ...

$$\text{ResourceWaiting}(p) \equiv (\exists r)(\text{ResourceRequested}(p) = r \wedge r \neq \text{nullresource} \wedge \text{Owner}(r) \neq p)$$

ResourceWaiting() must be *false* for all phases, implying that $\neg \text{ResourceWaiting}(p)$ must be *true* for all phases *p*. Therefore, the precondition for the acceptance of a 'resume-phase' event may be simplified to:

$$(RunningPhase=nullphase) \wedge (Phase(PhaseElect)=A) \wedge (Phase(PhaseElect) \neq nullphase) \\ \wedge (Mode(PhaseElect)=normal)$$

4. The postconditions associated with a 'request-phase' event include:

:release the resources acquired during the phase
 for r in $ResourcesHeld(A)$
 $Owner'(r)=\emptyset$
 $ResourcesHeld'(A)=\emptyset$

$ResourcesHeld$ is initially set to $\emptyset$ and is only altered by the postconditions accompanying the acceptance of a 'grant' event. Since it was shown in simplification 2, that there can be no 'grant' events, then these actions concerning $ResourcesHeld$ in the postconditions for a 'request-phase' have no effect. Furthermore, $Owner$ is initially set to $nullphase$ and is only changed as a result of the postconditions that accompany the acceptance of a 'grant' event. Consequently, all of the postconditions listed immediately above can be eliminated from the simplified DASA automaton without ill-effect. In fact, the state components $Owner$, $ResourcesHeld$, and $ResourceRequested$ can all be eliminated from the automaton as well.

5. Consider the precondition for the acceptance of an 'abort-phase' event:

$$(RunningPhase=nullphase) \wedge (Phase(PhaseElect)=A) \wedge (Mode(PhaseElect)=abort)$$

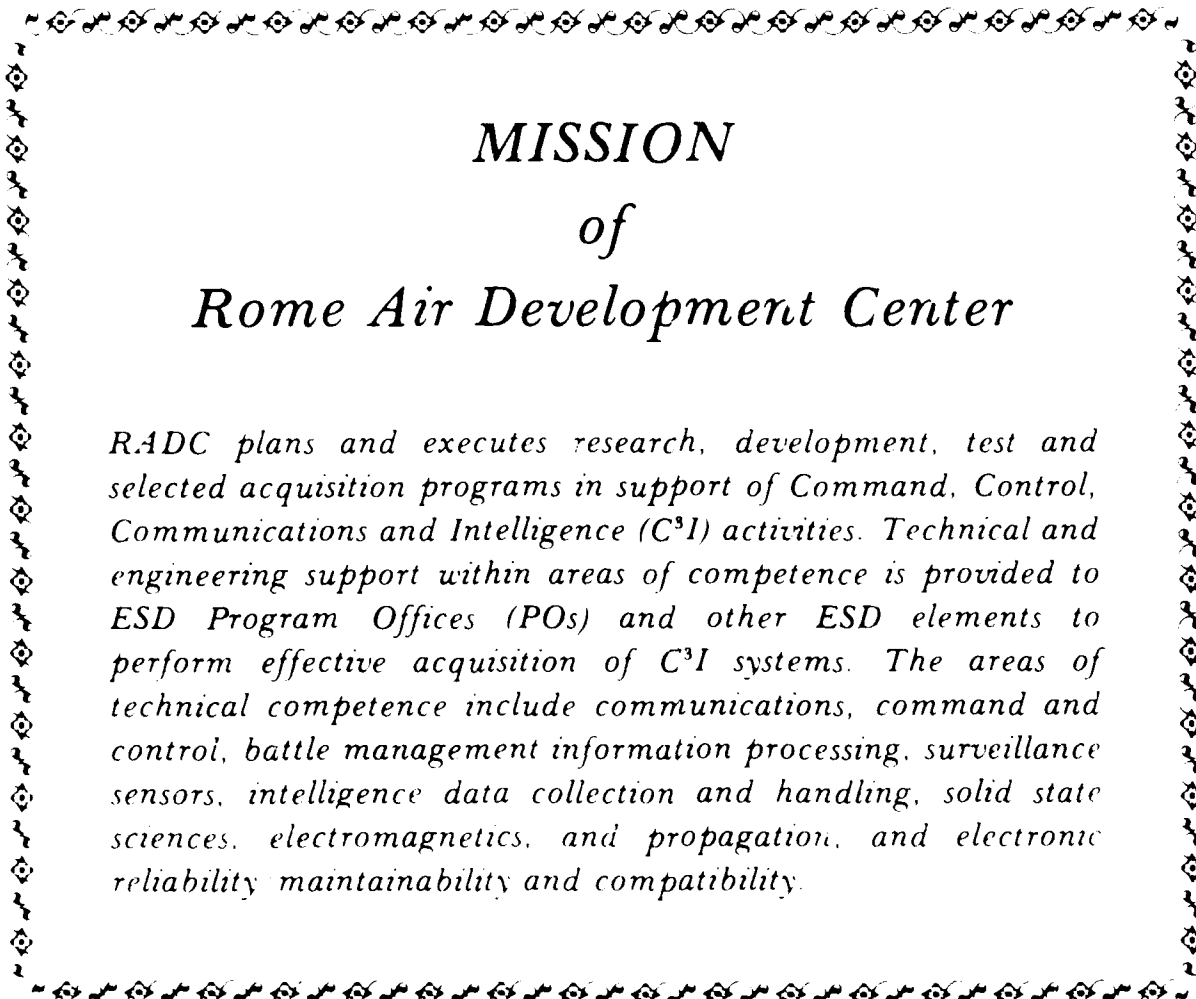
In particular, consider the conjunct $(Mode(PhaseElect)=abort)$. $PhaseElect$ always receives its value as a result of the following evaluation:

$$PhaseElect' = SelectPhase(PhaseList')$$

and ...

$$SelectPhase(P) = \\ pickone(mustfinishby(DL_{first}(pmplist), P_{scheduled}(\{p1, p2, p3\}))), \\ \text{where} \\ pmplist = tobescheduled(P_{scheduled}(\{p1, p2, p3\})) \\ \\ pickone(PMP) = \\ \begin{array}{ll} \langle normal, nullphase \rangle, & \text{if } PMP = \emptyset \\ \langle normal, p \rangle \mid \langle normal, p \rangle \in PMP, & \text{otherwise} \end{array}$$

Under no circumstances will this return a phase/mode pair with a mode indicating abort. Therefore, the conjunct $(Mode(PhaseElect)=abort)$ will always be *false* and the entire precondition is always *false*. Consequently, the entire 'abort-phase' portion of the DASA automaton may be omitted in the simplified version.



MISSION of Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability maintainability and compatibility.