The design of an adaptive, intelligent operating system scheduler
by Terence Lammers

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University

Abstract:
This paper describes the design and implementation of an adaptive, intelligent operating system scheduler, based on machine-learning of heuristics. This scheduler is termed the adaptive scheduler controller (ASC). The question which the project seeks to answer is whether expert system technology, specifically the technique of machine-learning of heuristics developed by Waterman (1970), can be successfully used to create an adaptive scheduler. Such an adaptive scheduler could be expected to operate in an environment which is not completely determined. The procedure for machine-learning of heuristics is as follows. The ASC makes a series of decisions and then observes the behavior of the system as determined by these decisions. From this feedback the ASC deduces what the correct decisions should have been and compares the correct decisions to the ones it actually made. For each incorrect decision the ASC will modify its set of production rules, or heuristics, so that the incorrect decision will not be made in the future. The results are that the ASC was able to adapt successfully to changes in its environment and to its own behavior. It learned rules to correctly assign priorities to processes even though the behavior of these processes changed. It also was able to recover from errors which it made in assigning priorities. One may conclude that the techniques of machine-learning of heuristics can be used to create an adaptive scheduler. This technique may find wider applications to systems programming problems in which the system is required to make complex decisions in a variable environment.

THE DESIGN OF AN ADAPTIVE, INTELLIGENT OPERATING SYSTEM SCHEDULER

by

Terence Lammers

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

June 1985

APPROVAL

of a thesis submitted by

Terence Lee Lammers

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

May 30, 1985
Date

Nicholas J. Freyland
Chairperson, Graduate Committee

Approved for the Major Department

June 5th 1985
Date

J. Dunbigh Stanley
Head, Major Department

Approved for the College of Graduate Studies

24 June 1985
Date

W. S. Malone
Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his/her absence, by the Director of Libraries when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Glenna Lammers

Date 21-JUNE-85

TABLE OF CONTENTS

TABLE OF CONTENTS--Continued

TABLE OF CONTENTS--Continued

TABLE OF CONTENTS--Continued

LIST OF TABLES

ABSTRACT

This paper describes the design and implementation of an adaptive, intelligent operating system scheduler, based on machine-learning of heuristics. This scheduler is termed the adaptive scheduler controller (ASC). The question which the project seeks to answer is whether expert system technology, specifically the technique of machine-learning of heuristics developed by Waterman (1970), can be successfully used to create an adaptive scheduler. Such an adaptive scheduler could be expected to operate in an environment which is not completely determined. The procedure for machine-learning of heuristics is as follows. The ASC makes a series of decisions and then observes the behavior of the system as determined by these decisions. From this feedback the ASC deduces what the correct decisions should have been and compares the correct decisions to the ones it actually made. For each incorrect decision the ASC will modify its set of production rules, or heuristics, so that the incorrect decision will not be made in the future. The results are that the ASC was able to adapt successfully to changes in its environment and to its own behavior. It learned rules to correctly assign priorities to processes even though the behavior of these processes changed. It also was able to recover from errors which it made in assigning priorities. One may conclude that the techniques of machine-learning of heuristics can be used to create an adaptive scheduler. This technique may find wider applications to systems programming problems in which the system is required to make complex decisions in a variable environment.

CHAPTER 1

INTRODUCTION

Purpose

The purpose of this paper is to describe the design and implementation of an adaptive, intelligent operating system scheduler, based on expert system technology. This scheduler is termed the adaptive scheduler controller (ASC).

Statement of the Problem

The question which this project seeks to answer is whether expert system technology, specifically the techniques of machine-learning of heuristics developed by Waterman (1970), can be used to create an adaptive scheduler. Such an adaptive scheduler could be expected to operate successfully in an environment which is not completely determined. One might ask why this technology is being applied to the problem of scheduling processes by an operating system. Adaptive algorithms and adaptive schedulers are known which optimize system performance based on numerical computations. The best that an adaptive scheduler based on Waterman's technology or any expert system technology could probably hope to do would be to match the performance of such an algorithm, and the expert system would probably require more overhead, since it would need to store and process its knowledge base.

The reason for the application of expert system technology to scheduling is that if it is possible to capture the adaptive, learning aspect of Waterman's technology, even if only in an area of the operating system for which computational algorithms are known, then it may be possible to use this technology in areas of the operating system where computational algorithms are not known, or where the cost of executing such an algorithm is prohibitively high. For example, Blevins and Ramamoorthy (1976) describe a dynamically adaptive operating system (DAOS) which is to optimize the performance of a computer system in real time. They indicate that heuristic techniques may be necessary to perform the optimization.

> The real-time constraints of the DAOS influence the modification step in numerous ways. Obviously, each optimization technique must generate solutions quickly enough for real-time usage. Secondly, the techniques must possess a high ratio of accuracy to cost. As a result, frequently only heuristic techniques may meet the constraints. The computation time of the exact solution may be clearly infeasible or the error component of the input data may degrade the advantage of the exact solution.

It is just heuristics and the machine-learning of heuristics which lie at the core of the ASC. This is the essence of the adaptability inherent in the ASC. As an operating system increases in size and complexity and as the problems which it attempts to solve increase in complexity, then the cost of computing exact solutions will also increase and the use of heuristics and the machine-learning of these heuristics will become more attractive and may emerge as the only way to handle some of the most complex problems.

One might wonder in general if expert systems can succeed in reducing computation costs. One example of possible savings is an

expert system presented by Feigenbaum (1984) which interprets ocean sounds in a very noisy environment. As Feigenbaum points out, doing this with statistical methods would require the power of more than one supercomputer, if it could be done at all. The expert system interpreted sounds at levels equal to or better than human performance and it required one hundred to a thousand times less computing than conventional numeric methods.

The ASC is just one step toward the creation of an adaptive operating system based on heuristics and machine-learning. An example of such an adaptive operating system is the high-level operating system for a distributed computing system, described by Enslow (1978), which must allocate resources without complete, accurate, or timely information about the state of all the nodes in the system. For such a system the ability to learn and adapt might be critical to its success.

The adaptive scheduler controller presented in this thesis, then, is seen as a test case for adaptive technology in operating systems and as a possible first step toward a complete, intelligent system to control an entire operating system (OS). This complete control system is termed the expert system operating system (ESOS). ESOS is seen as a member of a class of intelligent, adaptive control systems which control other systems. In practical terms, an intelligent system, or expert system, controlling an OS may be able to increase system throughput by more flexibly managing system resources than a conventional OS can. It could also make the system more user-friendly and automate the system manager's task.

Choice of the Scheduler as a Test Case

There are several areas in the OS which could be chosen as a test case for adaptive control by an expert system, such as the memory manager or the command language interface. The process scheduler has been chosen as the test case because it is a small, fairly well-defined problem. If a successful adaptive scheduler can be created using machine-learning techniques, then perhaps a complete, adaptive operating system could be built using the same methods.

Literature Review

It is the point of view of this thesis that there are two approaches to creating an adaptive OS or scheduler; the standard technique uses numerical methods. Some system parameters are measured and this information is used to compute the optimum behavior of the system. The system's behavior is modified to reach this optimum behavior. All the research encountered (Badel:1975, Blevins:1976, Geck:1979, Ishikawa:1982, and Potier:1976) takes this approach. One article (Sakamura:1979) describes a model of automatic OS tuning which includes an analyzer, a data base for learning, and the learning of tuning behavior; however, the learning is not based on expert system technology, but rather on numerical methods. In this thesis the expert system approach to creating an adaptive scheduler is examined.

There seems to be little research on the application of expert system technology directly to operating systems or schedulers. While the so-called Fifth Generation project (Feigenbaum:1984) relies heavily

on expert system technology, it is not clear whether an adaptive operating system based on this technology is planned or not. However, some research has been done in other systems programming areas. For example, the designers of a compiler compiler (Leverett: 1980) used a knowledge-based methodology, a technique borrowed from AI, to handle optimizations. They also used heuristic search methods, another AI technique, to generate code. An OS presents an interface to the user and AI techniques can be used to make this interface much easier to handle (Hayes:1981). Digital Equipment Corporation has also developed an expert system, XCON, which configures VAX systems (Kraft:1984). Like an OS, a data base management system is concerned with managing resources, improving response time, and providing a convenient user interface. AI techniques have been used to make it easier for the user to control the data base system, to increase its efficiency, and to extend its capabilities (Barr:1982).

Outline of the Thesis

This thesis is organized as follows. Chapter 2 introduces part of the XINU operating system (Comer: 1985). XINU's process scheduling is the basis of the simulation which drives the ASC. Chapter 3 introduces expert systems. A simple example of an expert system is given and a simple, introductory expert system scheduler will be presented. Chapter 4 describes the simulation which drives the ASC. Chapter 5 describes the design and implementation of the ASC. Chapter 6 presents the conclusions of the thesis.

CHAPTER 2

THE XINU FOUNDATION

Introduction

The ASC is to be a separate module built on the foundation of an existing OS scheduler. The existing scheduler calls the ASC to perform tasks, such as setting priorities, which are beyond the capabilities of this OS scheduler. The scheduler for the XINU operating system has been chosen as the foundation for the ASC. This scheduler is also the model for the simulation which drives the ASC. The XINU scheduler is a good choice because XINU is a small, simple, yet complete OS. A description of the XINU scheduler is given below together with descriptions of some other XINU modules which are part of its environment.

The History of XINU

The first description of XINU was published in 1984. It is a noncommercial system done as an academic project by D. Comer and a group of graduate students at Purdue University. Some of its ideas go back to the UNIX system developed at Bell Laboratories (Ritchie:1974). Its layered approach is well-known and has been used for some time. XINU is a small system designed to run on a microcomputer. Even though it is a small system, it contains all the major components of any OS: memory management, process management, process coordination

and synchronization, interprocess communication, real-time clock management, device drivers, intermachine communication and a file system. XINU was originally run on a VAX and an LSI-11/02. Later versions were developed for an Intel 8086 and a Motorola 68000.

XINU Process Manager

Process management under XINU consists of three major components: (1) scheduling and context switching; (2) a software layer on top of the scheduling and context switching layer which contains procedures to suspend, resume, create and kill processes; (3) a real-time clock manager. Each of the three components is discussed below.

XINU Scheduler

The XINU scheduler selects the ready process with the highest priority from the ready list and makes it the current process, that is, the executing process. A procedure, ready, places processes on the ready list. The scheduler accesses the process table to find the process with the highest priority. The process table is a data structure containing an entry for each process in the system. A process's entry contains all the necessary information to identify a process and to restart a process after it has been stopped. Priorities are set by users. All processes which have the same priority are scheduled round-robin. Round-robin scheduling uses a quantum and clock interrupts; these are discussed below. The scheduling procedure calls a context switching procedure to save the current registers and restore those of the process starting.

Other Scheduler Services

A layer above the scheduler provides suspend, resume, create and kill services. A suspended process is waiting for some restart condition and is not eligible to contend for the CPU. The suspend procedure receives the identification of the process to be suspended, verifies that the process is ready or current, removes it from the ready list, and marks the process as suspended in its process table entry. The resume procedure reverses a suspension by marking the process as ready and placing it back on the ready list. A new, independent process is created by a call to the procedure create. Create locates and initializes a free entry in the process table for the new process. Create also allocates stack space for the new process and writes values into this stack to allow the new process to begin executing. Create also fills in the register save area of the new process's process table entry so that the context switching procedure can start the new process. The kill procedure stops a process, if it is executing, and removes it from the system by clearing its entry in the process table.

XINU Real-time Clock Manager

A real-time clock manager is necessary to implement the round-robin scheduling algorithm, because each process of equal priority is guaranteed the same quantum of CPU service. At each context switch the clock manager schedules a preemption event. The event takes place in quantum clock ticks. A clock interrupt occurs at every clock tick and the current process's quantum is reduced by one tick. When the

process's quantum is zero the scheduler is called. The clock manager also schedules wakeup events for processes which request a timed delay and then suspend themselves.

Process Coordination

Processes need to coordinate with each other to synchronize their actions and to share resources. For example, if one process is producing values and another process is consuming them, then the two must synchronize their actions so that every value produced is received by the consumer. Also, if one process is updating a shared data structure then it must, as usual, be guaranteed sole access to that data. Process coordination is considered part of the scheduling environment because synchronization often means that a process is waiting for some signal and while it is waiting the scheduler will ignore it.

XINU Process Coordination

XINU implements process coordination with counting semaphores. Semaphores are simply integer counters. A process calls wait(semaphore) to decrement the counter and signal(semaphore) to increment it. If the semaphore becomes negative the process is placed in a waiting queue associated with this semaphore. A waiting process does not use the CPU. XINU maintains a table of semaphores. A process may request a number of semaphores, use them, and then release them.

Interprocess Communication

Processes often need to communicate with each other. XINU uses the technique of message passing to allow this communication. One process sends a message directly to another and not to a mailbox. Messages are limited to one word and if more than one message is sent to a process only the first to arrive is received. When a process is in the receiving state, waiting for a message, it does not contend for the CPU.

XINU Message Passing

XINU provides three procedures, receive, recvclr, and send to implement message passing. A process, wishing to obtain a message sent to it, calls receive. Receive checks the message flag associated with the process's message field. If there is a message, receive returns it to the process. If there is no message, receive puts the process into the waiting state and calls the scheduler to dispatch another process. Procedure recvclr is similar to receive except that it does not wait for a message to arrive. If there is a message, recvclr returns it; if there is none it returns OK. Procedure send, when called by a process, checks a flag in the receiver's process table entry which indicates if a message is already present. If there is one, send just returns to the process which called it. If there is no message, send sets the flag, deposits its message, and if there is a process waiting for the message, moves it to the ready list. Message passing is part of the scheduling environment in that processes move to and from the ready list.

CHAPTER 3

NONADAPTIVE SCHEDULER CONTROLLER

Introduction

The purpose of this chapter is to introduce expert systems and to describe a simple scheduler controller designed using expert system technology. This nonadaptive scheduler controller should only be considered as an example to help the reader understand the adaptive scheduler controller described in a later chapter. Some of the basic concepts of expert systems technology will be introduced and then the design of the nonadaptive scheduler controller will be presented.

Introduction to Expert Systems

An expert system is a program which has a built-in knowledge base which allows it to operate at a level of skill at some specific task. This knowledge base is usually large, difficult to define completely, and changeable with time. Facts may have to be added and deleted. Because the knowledge base may be large, it is important to represent it with generalizations. If each fact were represented individually, too much storage would be used. Even though the knowledge base may be difficult to define completely, it should still be usable in many situations. The changeable nature of knowledge requires that the knowledge base must be easy to update.

Design of the Knowledge Base

The design of an expert system is dominated by the necessity of dealing with large amounts of knowledge specific to a certain domain or task. This knowledge may be isolated facts, established procedures, educated guesses and global strategies. There are several methods of representing this knowledge base: (1) predicate logic, (2) procedural representation, (3) semantic nets, and (4) production systems. For expert systems the most common method is a system of productions and this method will be described here.

Transfer of Knowledge

Since expert systems usually require large knowledge bases, it is important to find effective means of transferring knowledge from human experts to the expert system, since human experts are usually the sources of the knowledge. Often this process involves the program's designer talking to human experts about their expertise and then coding this information. This transfer is a bottleneck which can slow down the creation of the program.

Maintenance of the Knowledge Base

After an expert system has acquired the necessary knowledge to perform its task, it should be able to maintain and expand its knowledge base as human experts do. That is, expertise is somewhat volatile. New facts are discovered and new procedures are developed. The expert system should be able to assimilate new information and to delete information which has been found invalid. Thus the knowledge

base is built up incrementally. This incremental nature of the knowledge base is one reason why production systems are often used; new productions can be added to or deleted from the system without restructuring it.

Production Systems

A production system is based on condition-action pairs which are termed production rules or productions. If the condition is fulfilled then the action is performed. The system is made up of these condition-action pairs, formed into a rule base, a context or context list which contains information about the current problem being solved, and an interpreter which controls the production system. This production system is the foundation of most expert systems.

Productions. The productions have a left-hand side, or condition part, which must be satisfied for the right-hand side, or action part, to be activated. A typical production might be "IF IT IS 10:00, THEN GO TO CLASS". An expert system might have several hundred such productions in its rule base.

Context List. The context list is a list or buffer which provides focus for the the production system. It contains the information or symbols which must match the left-hand side of a production for it to be activated. From the example above, the information that it is 10:00 must be in the context for the action "GO TO CLASS" to be performed. The context can be modified by one production so that it is

able to activate other productions. In this way chains of activations may be created.

Interpreter. The third major part of a production system is the interpreter. The interpreter searches the rule base to determine what action to take next. It matches the tokens on the context list with the conditions in the rule base. If more than one rule is applicable then the interpreter must resolve the conflict. After conflict resolution, the remaining, or lowest numbered production's activity is performed and its results are added to the context list.

Expert System Example

An example of a production system to identify food items (Barr: 1981) is given below. The productions are:

P1: IF ON-CL green THEN PUT-ON-CL produce.

P2: IF ON-CL packed in small container
THEN PUT-ON-CL delicacy.

P3: IF ON-CL refrigerated OR ON-CL produce
THEN PUT-ON-CL perishable.

P4: IF ON-CL weighs 15 lbs AND ON-CL inexpensive AND
NOT ON-CL perishable THEN PUT-ON-CL staple.

P5: IF ON-CL perishable AND ON-CL weighs 15 lbs
THEN PUT-ON-CL turkey.

P6: IF ON-CL weighs 15 lbs AND ON-CL produce
THEN PUT-ON-CL watermelon.

The context is just a list of symbols termed the context list (CL). The interpreter is:

1: Find all productions whose condition parts are true and make them applicable.

2: If more than one production is applicable, then deactivate any production whose action adds a duplicate symbol to the CL.

3: Perform the action of the lowest numbered (or only) applicable production. If no productions are applicable, then quit.

4: Reset the applicability of all productions and return to 1.

Example System Execution

The action of the above example system is as follows. Suppose that the context list contains two initial symbols, "green" and "weighs 15 lbs", then on the first cycle the interpreter matches P1. Since there is only one applicable production, no conflict resolution is necessary. The right hand side of P1 is executed and the symbol "produce" is put on the context list. The interpreter cycles until it can find no productions which do not put a symbol which is not a duplicate onto the context list. When this occurs, the interpreter quits and the result is the token most recently put onto the context list. Problems with the system can be fixed by adding rules, modifying rules, changing the order of the rules, or deleting some rules.

Design of the Nonadaptive Scheduler Controller

The rest of this chapter will describe the design of a nonadaptive scheduler controller based on a production system to give the reader some insight into the application of the expert system technology described above to the problem of designing a scheduler controller. This scheduler controller's significance is as an example only. Since it does not incorporate adaptability, the primary goal of this thesis.

The Knowledge Base

As noted above, the design of an expert system is dominated by a knowledge base. The knowledge base must be derived from a human expert and it must be abstracted and reduced to a set of production rules. The context list and an interpreter must be developed. In the case of the expert system to control the scheduler, there are no human experts to transfer the knowledge base. However, a knowledge base can be derived from the scheduling algorithms which have been implemented for several OS's. These algorithms can be abstracted and formed into a rule base for the scheduler expert system. Some of the best known scheduling algorithms (Deitel:1983 and Brinch Hansen:1973) first-in-first-out (FIFO), shortest-job-first (SJF), highest-response-ratio-next (HRN), round-robin (RR), shortest-remaining-time (SRT), and multilevel feedback queue (MFQ) are examined briefly below.

FIFO. FIFO is the simplest scheduling algorithm in that a process's priority depends only on its time of arrival to the ready queue. Once a process has been dispatched to the processor, it is not

preempted; it runs to completion no matter how long it executes. One long job can keep many shorter jobs waiting. Thus, FIFO does not maximize throughput in the sense that it does not service the maximum number of processes in unit time. FIFO is really only suitable for batch systems.

SJF. SJF dispatches the process with the smallest estimated run-time-to-completion next. SJF favors short processes over long ones and it would maximize the number of jobs serviced per unit time. The difficulty inherent in SJF is that it is difficult to estimate accurately the running time of a job unless it has been run several times already. Since SJF is nonpreemptive, once a process has the processor it runs to completion no matter what its estimated running time.

HRN. The highest-response-ratio-next scheduling algorithm attempts to be fair to both short jobs and long jobs. A process's priority is calculated by adding the time it has spent waiting to its service time and dividing this sum by its service time. Thus, if a processor's service time is small its priority will be high, but a long job's waiting time will increase until its priority too will become large and it will be dispatched. Since HRN is nonpreemptive, a process monopolizes the processor until it completes. HRN is adaptive in the sense that a process's priority changes to reflect different circumstances.

RR. Many interactive systems use the round-robin (RR) algorithm because it guarantees a reasonable response time. A process enters

the back of the ready queue and is dispatched when it reaches the head. A process is only allowed to execute for a certain time period called a quantum. Once this quantum is exhausted the process is preempted and placed at the back of the ready queue.

SRT. Processes with the shortest estimated time to completion are dispatched first by the SRT algorithm. After a process is given the processor it is not preempted unless a new process with a shorter estimated run-time arrives to the ready queue. Since SRT favors short jobs, it would tend to maximize throughput. However, to perform well, SRT needs accurate estimates of run-times. Users generally are not be able to give accurate estimates for most jobs, so the system would have to estimate their time from past performance.

MFQ. Perhaps the most interesting scheduling algorithm is the system of multilevel feedback queues, because it identifies a job from its behavior and services the job according to this behavior. In other words it adapts its service to a job. When a process enters the system it is placed in the highest level queue and is given a quantum. If the process gives up the CPU before its quantum has expired then it remains in this queue. If the process uses up its quantum, then it is moved to a lower queue where it will only receive service if there are no jobs waiting in the higher queues. As a process continues to exhaust its quantum at each level it moves down the system of queues until it reaches the lowest queue where it executes until it completes or until its behavior changes. In any of the lower queues if a process does not exhaust its quantum then it can move up a level in the system.

Since short jobs and I/O bound jobs would probably finish or block before they had used up their quantum, they would remain in the highest level queue and receive the best service. This would maximize throughput and the utilization of I/O devices.

Objectives of Scheduling Algorithms

From the description of the scheduling algorithms, it can be seen that these algorithms attempt to satisfy a number of different objectives. HRN attempts to be fair to both long and short jobs. SJF favors short jobs to maximize throughput. RR tries to guarantee a reasonable response to interactive users. A system designer must determine what the objectives of the system being designed are and create a mechanism to meet them. It seems that, unless one is designing a real-time system, an important objective would be to maximize throughput by servicing the maximum number of jobs per unit time. This objective requires that the objective of trying to be fair to all types of jobs be ignored because the best way of maximizing throughput is to favor short jobs. Another important objective is to obtain good I/O device utilization by favoring I/O bound jobs. This will also tend to maximize throughput, since an I/O bound job will block before it uses up its quantum and another process may be run. Since in an interactive system user response time is important, an objective for such a system would be to favor interactive jobs to guarantee an acceptable response time. The nonadaptive scheduler to be designed below has the objectives of maximizing throughput, maximizing utilization of I/O devices, and guaranteeing reasonable response to interactive users.

Abstracting Objectives

One would like to design the nonadaptive scheduler's knowledge base by abstracting the above objectives and using the abstraction to create the productions. To abstract the scheduler's objectives one might submit them to feature analysis. A job which uses little CPU time would be marked with the feature short. Its opposite, a job which uses a great deal of CPU time, would be marked with the absence of the short feature. A job which is marked short, I/O bound, and interactive would satisfy all of the scheduler's objectives and should receive the highest priority. Carrying out this feature analysis one arrives at the following table:

Features of Scheduler Objectives

Priority	Interactive	Short	I/O Bound
8	+	+	+
7	+	-	+
6	+	+	-
5	+	-	-
4	-	+	+
3	-	-	+
2	-	+	-
1	-	-	-

Given this table, which abstracts the scheduler's objectives, one can fairly easily design the production system to implement them.

Productions for the Scheduler Controller

From the table above and the example of the productions given earlier one can design the following productions to set a process's priority.

1: IF ON-CL priority then PUT-ON-CL modetask.

2: IF ON-CL modetask THEN PERFORM modetask.

3: IF ON-CL batch THEN PUT-ON-CL lengthtask.

4: IF ON-CL interactive THEN PUT-ON-CL lengthtask.

5: IF ON-CL lengthtask THEN PERFORM lengthtask.

6: IF ON-CL short THEN PUT-ON-CL boundtask.

7: IF ON-CL long THEN PUT-ON-CL boundtask.

8: IF ON-CL boundtask THEN PERFORM boundtask.

9: IF ON-CL batch AND ON-CL long AND ON-CL cpubound
THEN PUT-ON-CL priority 1.

10: IF ON-CL batch AND ON-CL short AND ON-CL cpubound
THEN PUT-ON-CL priority 2.

11: IF ON-CL batch AND ON-CL long AND ON-CL iobound
THEN PUT-ON-CL priority 3.

12: IF ON-CL batch AND ON-CL short AND ON-CL iobound
THEN PUT-ON-CL priority 4.

13: IF ON-CL interactive AND ON-CL long
AND ON-CL cpubound THEN PUT-ON-CL priority 5.

14: IF ON-CL interactive AND ON-CL short
AND ON-CL cpubound THEN PUT-ON-CL priority 6.

15: IF ON-CL interactive AND ON-CL long
AND ON-CL iobound THEN PUT-ON-CL priority 7.

16: IF ON-CL interactive AND ON-CL short
AND ON-CL iobound THEN PUT-ON-CL priority 8.

Most of these productions directly place a symbol on the context list; others, like number 8, perform a subtask to determine the appropriate symbol to put on the context list. For example, production 6 places

the symbol, "boundtask", on the context list. Once there, production 8 finds it and performs the subtask, "boundtask". This subtask determines if the process is I/O bound or CPU bound and places the appropriate symbol on the context list.

Context List for the Scheduler Controller

A production system is made up of a set of productions, a context list and an interpreter. The context list is a data structure designed to hold a list of symbols. The symbols are added at one end only, termed the head, and never removed until the whole list is destroyed. The list must be searchable, but the direction of the search is not important. Therefore, the context can be designed as a linked list with a pointer to the head node and a pointer from each node to its successor.

Interpreter for the Scheduler Controller

The third major part of the production system is the interpreter. The productions can be visualized and designed as a data base; the context list is a data structure; the interpreter can be designed as a procedure which operates on the productions and the context list. The interpreter should be designed to be as independent as possible of the productions so that productions may be added or deleted without making changes to the interpreter necessary. Since the interpreter is a procedure, its design can be described in an algorithm. The algorithm is given below.

1: Compare each symbol on the context list to the condition part of each production and if there is a match, then mark the production as applicable.

2: If more than one production is marked as applicable then look at the symbol which each production puts on the context list and if it duplicates a symbol already on the list, then deactivate the production.

3: There still may be more than one production activated so to resolve the situation execute the lowest numbered production. If no productions are activated then quit.

4: Deactivate all productions and go to 1.

Execution of the Controller's Production System

The general sequence of operations is that the scheduler extracts a process from the ready list and calls the expert system or controller with a flag telling the controller that the task at hand is to set a process's priority. The controller calls the interpreter with the first and last production numbers which the interpreter will need to determine the process's priority. In addition the controller adds any necessary, initial symbols to the context list. The interpreter calls a procedure, conditionpart, to match the condition part of each production with the symbols on the context list. The procedure conditionpart calls a procedure oncl to match the condition parts with the symbols. After all the applicable conditions are activated, the interpreter resolves conflicts and then calls the procedure actionpart for the remaining activated production. Actionpart calls the procedure putoncl to place a symbol on the context list. The interpreter continues to execute until there are no applicable productions; control returns to the controller which sets the priority of the process and returns control to the scheduler.

Pseudocode for the Scheduler Controller

Pseudocode in an English-like high level language is given below for each of the procedures of the scheduler controller.

Scheduler

```
procedure scheduler
     while there are processes on ready list do
          call controller ( set priority )
          get the next process on the list
     end while
end scheduler
```

Controller

```
procedure controller ( requested action )
     case of requested action
          set priority :
               call putoncl ( initial symbol )
               call interpreter ( first
                    production, last production )
               remove priority from context list
               and assign it to the process
          next action :
                                        * * *
               * * *
     end case
end controller
```

Interpreter

```
procedure interpreter ( first production,
     last production )
     while there are active productions do
          for first production to last do
               call conditionpart ( production )
          end for
          if more than one production active then
               for each active production do
```

```
                    check to see if the symbol
                    to be placed on the
                    context list duplicates
                    a symbol already there
                    if symbol is duplicate then
                         deactivate ( the production )
               end for
          if there are active productions then
               call actionpart ( lowest
               production )
          else there are no active productions
     end while
end interpreter
```

Conditionpart

```
function conditionpart ( production number )
     case of production number
          production 1 : if oncl ( priority )
                              then match is true
               * * *
     end case
end conditionpart
```

Actionpart

```
procedure actionpart ( production number )
     case of production number
          production 1 : putoncl ( modetask )
               * * *
     end case
end actionpart
```

Oncl

```
function oncl ( symbol )
     while symbol on the context list do
          if symbol match on context list then
               oncl is true
               return
          end if
          get the next symbol
     end while
end oncl
```

Putoncl

```
procedure putoncl ( symbol )
     get a new node
     add symbol to the node
     link node to head of context list
end putoncl
```

Summary

In this chapter expert systems were discussed and an example was given. The main components of an expert system, the knowledge base, the productions which can be used to implement the knowledge base, the context list and the interpreter were presented. The design of a nonadaptive scheduler controller was presented and its procedures were given in pseudocode. This scheduler controller is nonadaptive in the sense that no matter how its environment changes the productions which the scheduler controller uses to set priorities do not change.

CHAPTER 4

SIMULATION TO DRIVE THE ASC

Introduction

After the ASC has been designed and implemented, it must be driven by a simulation to verify that the design is correct and to obtain some results. The simulation to drive the ASC is given below. First, the derivation of the interarrival time and the service time is explained. The elements of the simulation are presented and then the design and implementation of the simulation are described. The ASC driver simulates the arrival, execution and blocking of processes. It follows the XINU scheduling operation, although not in all details. The ASC itself is embedded in the simulation and operates as if it were being driven by a real OS scheduler.

Interarrival and Service Times

The two important input random variables to the simulation are the interarrival time and the service time. The interarrival time represents the time interval between the arrival of two jobs to be serviced by the OS. The service time is a job's execution time.

Interarrival Time

The mean, or expected, interarrival time for a typical time-sharing system is 23 seconds (Brinch Hansen:1973, Coffman:1966). The

Coffman article points out that the interarrival times do not fit an exponential distribution very well and that a better fit is provided by a hyperexponential distribution. Despite the better fit of the hyperexponential distribution, the exponential distribution is used here for the sake of simplicity. The average arrival rate is:

$$\lambda = 1 / E(\tau)$$

The expected interarrival time is $E(\tau)$. This relation was used to compute the average arrival rate and this result was used to convert random numbers from a uniform to an exponential distribution for the simulation.

Service Time

Brinch Hansen gives the average execution time as 1.19 min and references Rosin (1965) and Walter (1967) as the sources for this value. However, Rosin gives 1.9 min as the average service time per job and Walter does not give a single, average time. Walter does give a graph of execution times from which it is possible to derive some values to analyze. This analysis is given below.

Estimated Service Time. Forty execution times were derived from Walter's graph. A chi-square goodness-of-fit test on these execution times determined that they are from an exponential distribution. The equations for this test are 9.19 and 9.20 from Banks (1984). An estimated average service time of 2.15 min was calculated from Walter's data. This value is used in the simulation.

Unstable System

The average arrival rate calculated for the simulation is 0.00435. The average service rate is 0.000775. Since the arrival rate is greater than the service rate, the queuing system is unstable. The server, the CPU, will get further and further behind. The long-run average queue length is infinite. All long term performance measures such as the expected number of processes in the system will also become infinite and have no meaning. This instability is not significant for the present simulation of the ASC.

Basic Elements of the Simulation

There is a small number of basic elements which are necessary for the simulation to drive the ASC. These are: (1) a future event list (fel), (2) a clock, (3) a random number generator, (4) a simulation algorithm, and (5) procedures to gather statistics. Each of these will be discussed below.

Future Event List

The fel contains future events and the times at which these events will occur. A two-dimensional array will satisfy this requirement. It is loaded with an event's code and with the event's time. The size of the array is some worst case estimate of the possible number of pending events, but need not be too large, since as events occur they are removed from the fel.

Clock

A simulation clock contains the current event's occurrence time. The clock is updated to each event's time, as that event occurs. The clock is simply a real variable.

Random Number Generator

This simulation uses the UNIX Pascal random number generator which produces uniformly distributed random numbers from 0.0 to 1.0 with a mean of 0.5. Since the scheduling simulation requires numbers from two exponential distributions, these random numbers are produced by inversion from the uniform distribution by the following equation (Bratley:1983):

$$x = -\ln(1-u) / \lambda$$

Simulation Algorithm

The general simulation algorithm (Gordon:1978) is given below.

Scan the events to determine the next potential event.

Select the activity to cause the event.

Test if the potential event can be executed.

Change the state of the system to reflect the effects of the event.

Gather statistics for simulation output.

Gathering Statistics

To gather statistics one can install counters and variables at appropriate places in the simulation program. The gathered statistics

are analyzed and reported by a procedure called before the program exits.

Design and Implementation of the Simulation

If one considers OS scheduling as a system of states with transition paths between the states and at the same time considers the simulation algorithm given above, then one may fairly easily design a simulation program which follows both this system of states and the simulation algorithm. The state system is a simplification of XINU's scheduler which was described in Chapter 2. The state system of the scheduling process is discussed below, and this is followed by a description of the simulation program.

State System

The state system of the scheduling process can be described as a number of states and the paths which represent transitions between states. An event which occurs while a job is in one state will cause it to take a path to another state. If more than one path is available, a certain event will completely determine which path is to be taken. Each of six possible events will be discussed in turn, together with the state to which each moves a job.

Arrival Event. The arrival event brings a new process into the system. The process is taken to the ready state. This ready state represents the ready queue in the real OS.

Dispatch Event. A process which is in the ready state is moved to the run state by the dispatch event. This event represents the acquisition of the CPU by a process.

Clock Event. From the run state a process makes the transition back to the ready state when a clock event occurs. However, the process only takes this transition path when its quantum is exhausted.

Done Event. When a process is in the run state, it may complete its execution. In this case a done event occurs and the process is moved out of the system.

Block Event. Also, when a process is in the run state, it may make an I/O request. If the process makes such a request, the block event occurs and the process moves to the blocked state.

Complete I/O Event. After a process has reached the blocked state, it remains there until it has completed its I/O. When the I/O is finished, the complete I/O event moves the process back to the ready state.

Special Interrupt Events

The block event and clock events can be considered as special interrupts because they cause calls to the scheduler to set priorities and to the dispatcher to give the CPU to the process with the highest priority.

Design of the Scheduling Simulation

The overall design of the simulation follows the state system fairly closely. An event occurs and a procedure corresponding to the state determined by this event is called to handle the event. Each of these procedures will be described.

Simulation Algorithm

The simulation algorithm begins by initializing all variables, scheduling the first clock event, and scheduling the first arrival. After initialization, the main program loop takes over and runs until the end event occurs. The main loop calls a procedure to scan the fel to get the current event and time. The current event determines which event handler is called. When the main loop terminates, a procedure to collect, analyze and print statistics is called.

Scanfel

The scanfel procedure looks through the future event list and determines the event which is to occur next. The event to occur next is simply the event which has the minimum event time. The time and code corresponding to this event are removed from the fel and their space is freed. Scanfel then makes the time and event code available to the main loop.

Main

The main body of the program receives the current event and time from scanfel and calls the procedure which handles this particular

event. After the called procedure returns to the main loop, scanfel is called again to get the next event.

Clock Event Handler

The procedure which handles the clock event has two main tasks. First, it schedules the next clock event by calling the procedure schedule event. After this, the clock event procedure subtracts one from the current process's quantum. If the quantum becomes zero, the scheduler and dispatcher procedures are called to find and dispatch the next current process.

Block Event Handler

The block event handler determines the length of the I/O request which a process has made. The handler schedules an I/O completion event at the end of the request. The process which made the request is marked as blocked.

Complete I/O Event Handler

This procedure simply removes the process which has completed its I/O from the blocked state.

Done Event Handler

This handler gathers statistics, such as the time which the process spent in the system. After these statistics have been recorded, the process is removed from the system.

Arrival Event Handler

Each time a new process comes into the system, another arrival is scheduled. Then the new process is initialized. That is, it is given an identification number, a time stamp, and several other values. Finally, the new process is linked into the ready queue.

End Event Handler

This procedure terminates the main loop and transfers control to the statistics reporting procedure.

Scheduler

For each non-blocked process on the ready queue whose priority has not already been set, the scheduler procedure calls the scheduler controller to determine this process's priority. The scheduler ignores the null process which is always available to use any extra CPU time.

Dispatcher

The dispatcher looks through the ready queue for the process with the highest priority. This process is given the CPU. Before the process is actually dispatched, the dispatcher determines if the process will make an I/O request during its quantum. If it will, a block event is scheduled. The dispatcher also determines if the current process will complete during its quantum. If the process will complete, a done event is scheduled.

Schedule Event

The schedule event procedure accepts an event code and an event time and loads them into the fel. All events are scheduled relative to the current time.

Summary

The simulation to drive the ASC was described above. This simulation is important to the ASC because it models the scheduling environment to which the ASC must adapt. Also, by driving the ASC, the simulation makes it possible to obtain results from the ASC's operation.

CHAPTER 5

ADAPTIVE SCHEDULER CONTROLLER

Introduction

Chapter 3 described the design and implementation of a simple expert system to control the scheduling of processes for execution by an operating system, but this scheduler controller was not adaptive. The goal of this project is to develop an expert system which will adapt to its scheduling environment. Adaptation for the purposes of this investigation is considered to be learning and modifying a set of production rules which allow the scheduler controller to assign priorities and quanta in such a way that scheduling is maximized. The description of the ASC will include a definition of the problem, a solution (an algorithm), and the results of implementing and executing the algorithm.

Problem Definition

The problem is to design and implement an adaptive scheduler. To create this design it is first necessary to define what an adaptive scheduler is. One way to define such a scheduler is to describe the algorithm which the scheduler is to perform. This approach has the benefit that the defining algorithm is the first step to creating the design of the scheduler. The algorithm can be developed in greater and

greater detail to complete the design. Thus the solution of the design problem will begin, in top-down fashion, with a simple algorithm. Later this algorithm will be developed in detail. Of course, the logical conclusion to this approach is the actual code to implement the design which is given in Appendix A.

Simple Algorithm

A simple algorithm for the adaptive scheduler controller is the following. First, it performs its task for a certain interval. At the end of the interval, the controller obtains some feedback. Then the controller evaluates its performance during the interval based on this feedback and a knowledge of its desired performance. It must modify its behavior if the feedback indicates that it is not performing well.

Central Problem

From a consideration of the above algorithm one sees that there are two important features. First, there is a two-phase system. There is a normal execution phase in which the ASC performs some routine task and there is an evaluation phase in which the ASC evaluates its behavior and modifies it as required. Also, one might feel intuitively that behavior modification is a more difficult task than the others in the algorithm and is indeed the central problem in the design. This intuition will be borne out by the complexity of the algorithm to implement behavior modification. In fact the problem is to create code which can modify itself. A virtual machine must be created which can execute a set of instructions and it must be able to

modify its instruction set. A more complete algorithm for the operation of the scheduler controller is given below.

Operation of the ASC

Operation During Normal Scheduling Interval

The scheduler controller schedules jobs until the end of the current normal scheduling interval. Jobs are scheduled by the existing production rule list and current state of each process. The program subvector (Waterman: 1970) contains the information necessary to perform the scheduling. A job's priority and quantum are set. The program subvector and the decision derived from this subvector are saved for use during the evaluation cycle. Performance information is collected in the performance variables.

Evaluation Cycle

At the end of the normal scheduling interval, the scheduler controller enters the evaluation cycle. The controller gets feedback by examining each of the decisions it made during the normal scheduling interval. It deduces if it made the correct decision based on the charateristics of the processes and the performance of the system. That is, the ASC gets feedback on its scheduling performance. If the ASC determines that it has made the correct decision, then it makes no changes to the productions and just continues by examining the next decision made during the scheduling interval. If the ASC deduces that it has made an incorrect decision, then it begins a training session which will lead to the modification or creation of some productions.

Once all the decisions have been evaluated and any necessary training sessions have been conducted, the ASC returns to the normal scheduling interval.

Decision Evaluation. The process of decision evaluation operates as follows. Each of the saved program subvectors and its associated decision are considered in turn. First, they are used together with the performance variables to set the predicates in the axioms. The axioms can be considered as general rules or common sense rules for scheduling. The axioms are described more fully later. Using these axioms the ASC deduces what decision should have been made in order to have maximized scheduling. By maximizing scheduling is meant maximizing throughput, reducing response time, reducing overhead, and reducing the size of the ready queue. If the decision which the controller should have made and the decision which it actually did make are not the same, then a training session is performed.

Training Session

The purpose of a training session is to modify the list of productions so that the correct decision, as deduced above, will be made. With this correct decision and information from a decision matrix (explained more fully later), the controller will modify an existing production rule or add a new one to the list if none is suitable for modification.

Completion of the Evaluation Cycle

After all the saved program subvectors have been considered, and the appropriate modifications have been made to the list of production rules, the evaluation cycle is over. The performance variables are stored for comparison during the next evaluation cycle. A new scheduling interval begins and the scheduler will send processes from the ready list to the ASC and these processes will have their priorities and quanta set according to the new production rules.

Summary

As can be seen from the problem definition given above, the problem is in two major parts. The first of these is the normal scheduling interval during which the productions remain fixed and are used only to set priorities and quanta. This in fact is the function carried out by the nonadaptive scheduler described in Chapter 3. The other major part of the problem is the evaluation cycle during which scheduling decisions are tested and productions are modified if they make incorrect decisions. The process of modifying them is termed the training session. Each of these major parts will be considered in more detail below.

Detailed Example: Normal Scheduling

The context for the ASC's normal scheduling interval is the following. A process which has been dispatched to the CPU is executing. After its quantum is exhausted, or if it performs an I/O operation, the process must give up the CPU and return to the ready list.

The scheduler then calls the ASC to set the priority and quantum of each unblocked job on the ready list. The scheduler selects the job with the highest priority and dispatches it. This continues until the ASC's scheduling interval is over and the evaluation cycle begins.

Basic Concepts

BF Rules. The first step in the process of setting priorities and quanta is to change the program subvector (PSV) into a symbolic subvector (SSV). The program subvector is a vector of program variables which represent the characteristics of the process whose priority is to be set and the state of the system. The SSV is a vector of symbols which have been derived from the PSV by means of the so-called bf rules. The bf rules, or heuristic definitions, define subranges of a program variable and assign a symbol to the SSV based on these subranges. An example of a bf rule is the following:

```
interactive --> mode   mode >= 1
batch --> mode         mode < 1
```

This rule says that if the current value of the variable mode in the PSV is greater than or equal to 1, then the symbol interactive is assigned to that variable in the SSV. If the current value of mode is less than 1, then the symbolic value batch is assigned to the SSV. Each element of the current PSV is matched against the right sides of all the bf rules. If a match occurs, the left side of the matching bf rule is used to assign symbolic values to the SSV.

AC Rules. After the bf rules have transformed the PSV into the SSV, the ac rules (action rules) are used to create a modified program subvector (MPSV) from the PSV. This MPSV is also termed the actual decision because it represents the final decision of the program to assign some priority or quantum. An example ac rule is given below.

(batch, *, *, *, *, *, *, *) --> (*, *, *, *, 2, *, *, *)

The meaning of this rule is that if the value of the mode variable in the SSV is batch, then assign the value 2 to the priority variable of the MPSV. An asterisk on the left side of the rule indicates that the value of that variable is not taken into account. An asterisk on the right side indicates that the value of the corresponding variable in the MPSV remains unchanged. The SSV is matched against the left side of each ac rule until a match is made. Once a match occurs the MPSV is modified as described by the right side of the matching ac rule and matching ends.

Modified Program Subvector. The MPSV created by the ac rules contains the priority and quantum for the process whose characteristics were presented to the scheduler controller as a PSV. Its priority and quantum are now extracted from the MPSV and entered into the process control block belonging to that process. The setting of priorities and quanta with the bf rules and ac rules is analogous to the process of setting priorities and quanta with the system of productions, context list and interpreter which is used by the nonadaptive scheduler controller.

Setting the Priority

Let us assume that the scheduler is operating during the normal execution cycle. The scheduler controller is setting priorities and quanta for jobs using the current productions. A certain process's characteristics have been extracted from its process control block (PCB) and entered into a PSV. The PSV might look like the following:

PSV

m	t	b	q	p	*	*	*
*	2	*	10	1	*	*	*

In this example, m is the mode variable which indicates if a process is interactive or batch; t is the time variable, indicating the number of quanta which the process has used; b, the bound variable, contains the number of I/O requests which the process has made; q contains the process's quantum; p holds its priority. If the value of a variable is unknown or irrelevant, it is marked with an asterisk. The other variables, marked with '*' are currently unused. Each element of this program subvector is matched against all right sides of the bf rules. When there is a match, the corresponding left side of that bf rule is used to assign values to the SSV. For example, the bf rule

```
short --> time  time <= 2
```

would create the following symbolic subvector given the PSV above.

SSV

m	t	b	q	p	*	*	*
*	short	*	*	*	*	*	*

The next step is to produce a modified program subvector. This is equivalent to making a decision. To do this the symbolic subvector is matched against all left sides of the action rules, from top to bottom, and when the first match is found, the values of the program subvector are modified as described by the right hand side of the matched rule to create the MPSV. Given the SSV above and the following ac rule,

(*, short, *, *, *, *, *, *) --> (*, *, *, 10, 2, *, *, *)

the modified program subvector which would be produced is:

MPSV

m	t	b	q	p	*	*	*
*	*	*	10	2	*	*	*

The assigned priority and quantum will be transferred to the job's PCB. This concludes the example of the normal scheduling interval. A detailed example of the evaluation cycle will be presented next.

Detailed Example: Evaluation Cycle

Introduction

Once the scheduling interval is over, the scheduler controller enters the evaluation cycle. This cycle has two phases. During the first phase the controller evaluates each decision which it made

during the scheduling interval. If the decision is determined to be acceptable then the controller makes no changes to the productions and continues to the evaluation of the next decision. If the decision is unacceptable, then the second phase is entered, the training session in which the controller modifies the productions so that this unacceptable decision will not be made in the future.

First Phase

Once the scheduling interval is over, the program begins retrieving the decisions which it made during this interval and evaluates each of them in turn. Each decision is made up of the SSV which led to this particular decision, the PSV which was used to create the SSV, and the MPSV. These are retrieved and used to set the predicates of the axioms. The predicates are also set with the performance variables. An example of setting predicates is given below. After setting the predicates the program deduces what the correct decision should have been for the given circumstances. If the correct decision and the actual decision do not match, then the program calls for a training session (second phase).

Second Phase

The general training procedure is as follows. The ASC must construct a training rule which will produce the correct decision under the given set of circumstances. It creates a left side for the training rule of variables which will match the SSV of the PSV which is being evaluated. A right side of operations and operands to modify the MPSV to the correct decision is also created. The bf rules are

modified, if necessary, so that they reflect the current PSV. New bf rules may be created if none can be found to modify. Finally, the ac rule which caused the incorrect decision is located and the training rule is inserted above this error-causing rule in the list of productions.

Detailed Example of Both Phases

Axioms. The first step in the evaluation cycle is the setting of the predicates in the axioms. These predicates are set based on the performance variables and the program subvector associated with the decision being tested. These axioms represent a knowledge base of general rules about scheduling. The actual axioms which the ASC includes are the following.

1. increase(throughput) --> max(scheduling)
2. setpriority(high) and priority(low) and time(short) --> increase(throughput)
3. setpriority(low) and priority(high) and time(long) --> increase(throughput)

The first axiom means that increasing throughput is a way to maximize scheduling. The second states that setting a process's priority high if its current priority is low and the process is short will increase throughput. A process is considered short if it has executed for only a small number of quanta. The third is the complement of the second. For example, if the decision presented above as an example were being tested, then the predicates would be set as follows:

priority(low) = true

time(short) = true

Any predicates which cannot be marked true or false are marked as 'unset'. The program can use them to try to make true a chain of reasoning to an acceptable action. In the second and third axioms above the element setpriority(high) would be marked as an action. If the program can reach this action, then it will consider that this action is the correct one for the given set of circumstances.

Deducing the Correct Decision. To deduce the correct decision which the ASC should have made under a given set of circumstances, the program must reach an action by a chain of reasoning. This action is considered to be the correct decision for these circumstances. The chain of reasoning is created by matching predicates to the right sides of axioms. If a match is made, then the program tries to reach an action by matching the predicates on the left side of the axiom to the right sides of other axioms until it reaches an action or until it cannot extend the chain. For example, the program would begin deducing a correct decision by first matching the predicate maximize(scheduling) against the right sides of all the axioms until a match is made. In the example, maximize (scheduling) would match the right side of the first axiom. There is only one predicate on the left side and it would have been marked 'unset'. The program would be able to continue its chain of reasoning by matching this predicate, increase(throughput), against the rest of the axioms. To prevent looping an axiom would be

marked as 'used' once it has been matched. The predicate, increase(throughput), matches the right side of the second axiom. Since the predicate, time(short), is true, the program adds this to its chain and continues. The next predicate, priority(low), is also true so the program continues. The last predicate is an action; the program has deduced that in the circumstances that a process is short and has a low priority the correct decision is to set the process's priority high.

Comparing Decisions. Having found the correct decision, the ASC must compare this with the decision which it actually made. This actual decision was stored during the normal scheduling interval. This comparison indicates whether a training session is required or not. If the correct decision is the same as the actual decision then no modifications are necessary to the ac rules. The actual decision, or MPSV, is given below.

MPSV

m	t	b	q	p	*	*	*
*	*	*	10	2	*	*	*

This decision is unacceptable because the program deduced that the correct decision is to set the process's priority high and this process was assigned a low priority. Since the decision is unacceptable, the program begins a training session.

Training Session.

Decision Matrix

The main purpose of the training session is to create a training rule which will make the correct decision. The first step in creating a training rule is to obtain the relevancy and justification information from the decision matrix. The ASC's decision matrix is below.

	variables				
	m	t	b	q	p
decision					
sph	interactive	small	large	small	small
spl	batch	large	small	large	large

The correct decision, sph (set priority high) or spl (set priority low), is used as an index into the matrix. From the matrix the program finds that all the variables are relevant. It must hypothesize that some are not actually relevant when the training rule is inserted into the list of productions. The justification information is that the decision is made because the variables have the values listed in the matrix. Next the program constructs a training rule with a left side and a right side. The acceptability information becomes the right side and the relevancy and justification become the left side.

Training Rule Creation

The training rule is created from the relevancy, justification, and acceptability information. The relevancy information is used to create the left side of the training rule. This left side is the part of the action rule against which the symbols in the SSV are matched.

The values of the relevant variables from the relevancy information are put into their positions in the left side of the training rule after being transformed into values compatible with the bf rules. The example below is of the left side of a training rule (TLS) created for the sph decision.

TLS

m	t	b	q	p
interactive	short	iobound	small	small

The right side of the training rule is formed from the acceptability information. This acceptability information is just the correct decision which was deduced above. For example, the correct decision deduced earlier is setpriority(high). This decision can be analyzed into an operation, set, an operand, priority, and a value for that operand, high. The operand from the decision becomes the variable on the right side of the training rule which is to be changed if an SSV matches the left side. In fact, the variable in the training rule is not changed, but the corresponding variable in the MPSV is altered. The operation from the decision becomes the operation in the right side to be performed on the operand. The value is the right hand side of the operation to be performed. For example, the correct decision can be interpreted as:

priority <-- high

In this case the operation set is equivalent to an assignment. The numeric value of high can be set to any convenient number. Thus the

right hand side of the training rule contains two subrules, one containing the operation (TASOPS) and the other the value of the right side of this operation (TRSOPRNDS).

TRSOPS

m	t	b	q	p	*	*	*
*	*	*	*	assgn	*	*	*

TRSOPRNDS

m	t	b	q	p	*	*	*
*	*	*	*	5	*	*	*

At this point the program has created a complete training rule which will assign a priority level of five to any process which has been identified as short and whose current priority is low.

Modification or Creation of BF Rules. After the program has created a training rule, it must modify a bf rule for each of the relevant variables based on the justification information derived for the training rule from the decision matrix. If no bf rule corresponding to one of the relevant variables exists, then the program must create one. A bf rule corresponding to the relevant variable is necessary because this bf rule assigns symbols to the SSV which will be matched against the left side of the training rule, when it becomes one of the productions. To modify a bf rule the program searches the variables of all the bf rules to find one which matches a variable in the training rule. If a matching variable is found then the boundary of the subranges of the variable which the bf rule determines is moved

to reflect the current situation. If no matching variable is found, then the program must create a bf rule. To create a bf rule the program extracts the relevant variable from the relevancy information and inserts this into the new rule. The boundary of the subranges of the new rule is the value of the relevant variable from the PSV. Each bf rule has an implied less than or equal conditional operator. If the value of the variable in the PSV which matches the variable in the bf rule is less than or equal to the boundary value, then the SSV is assigned one symbolic value, termed the true value. If the value in the PSV is greater than the boundary, then the SSV is assigned another value, termed the false value. The true value and the false value are part of the relevancy information and are incorporated into the new bf rule. An identification number for the training rule is also put into the new bf rule. The new or modified bf rule may change the value of the SSV derived from the PSV being evaluated. If the SSV is indeed changed, then this new SSV may trigger the correct decision. If the correct decision is made, then the program is done with the training session for the PSV it was evaluating and it may move on to the next PSV. If the SSV is not changed or the correct decision is not made, then the program must continue with the training session and insert the training rule into the set of productions.

Insertion of Training Rule. As the final step of the training session, the program must insert the training rule into the current set of ac rules. The strategy of modifying an existing ac rule to reflect the training information (acceptability, relevancy and

justification) rather than just inserting the training rule, is not pursued here. This will lead to some redundant rules and to slower convergence to a stable system. Not modifying ac rules considerably simplifies the algorithm and will not change the basic results. Therefore, this simplification is used in this demonstration system. The training rule is inserted into the action rule list immediately above the error-causing rule. The error-causing rule is found by dropping the SSV being evaluated through the ac rules and locating the one which causes the incorrect decision. The training rule is inserted into the list above this rule. At this point the training session is over and the program continues with the next PSV.

Results

The purpose of this project is to create an adaptive scheduler based on machine-learning of heuristics. Therefore, the adaptive behavior of the ASC will be examined. One typical program run began with one job entering the system. It was originally identified as batch, short and CPU bound. Since no productions had been learned yet, this job was assigned the default priority, 1, by the one inherent ac rule which matches any SSV. Before the first scheduling interval was over, the job's behavior had caused it to be labeled as batch, long, and I/O bound. During the first training session three ac rules were learned. The acronyms LS, RSOPS, and RSOPRNDS have the same meaning as TLS, TRSOPS, and TRSOPRNDS except that they indicate the subrules of a regular AC rule and not the subrules of a training rule.

LS

m	t	b	q	p
*	*	*	small	small
*	*	cpubound	small	small
*	short	cpubound	small	small

RSOPS

m	t	b	q	p
*	*	*	*	assgn
*	*	*	*	assgn
*	*	*	*	assgn

RSOPRNDS

m	t	b	q	p
*	*	*	*	5
*	*	*	*	5
*	*	*	*	5

These three rules failed to correctly assign a priority to the job. Since the process was batch and long, one would like to see the ASC assign it a low priority, despite the fact that it was also identified as I/O bound. Not only did the ASC not assign a low priority to the job, it created an unstable situation in that it alternately assigned a high priority to the job and then failed to catch the job at all on any of its ac rules so that the job was assigned the default priority by the inherent rule. This alternation continued for one entire scheduling interval. During this interval another process entered the system. It was identified as interactive, short, and CPU bound. It was correctly assigned a high priority. Another training session took place. The ASC learned another ac rule.

LS

m	t	b	q	p
batch	long	*	*	*

RSOPS

m	t	b	q	p
*	*	*	*	assgn

RSOPRNDS

m	t	b	q	p
*	*	*	*	2

This rule ended the alternation between the high and default priorities for the batch job. It was now correctly assigned the low priority, 2, and was correctly assigned this priority until the end of the simulation. Thus the ASC had adapted its behavior to the batch job, but had done so incorrectly. Nonetheless, it was able to learn from its incorrect behavior and to learn a succesful solution to the problem. This is exactly the kind of adaptive behavior which is the goal of this project. However, at the end of the second training session, mentioned above, the ASC began to incorrectly assign the interactive job the default priority. In fact the ASC began the same kind of alternation between the default priority and the high priority for the interactive job. This alternation continued until the next training session. During this session the ASC learned another ac rule.

LS

m	t	b	q	p
interactive	short	cpubound	small	*

RSOPS

m	t	b	q	p
*	*	*	*	assgn

RSOPRNDS

m	t	b	q	p
*	*	*	*	5

This rule did not end the alternation. The ASC continued to assign an incorrect priority to the interactive job half the time. Another training session took place. The ASC learned the following rule.

LS

m	t	b	q	p
interactive	short	*	small	*

RSOPS

m	t	b	q	p
*	*	*	*	assgn

RSOPRNDS

m	t	b	q	p
*	*	*	*	5

This rule correctly captured the interactive job and correctly assigned it the high priority, ending the alternation. Here again the important fact is that the ASC was able to recover from its mistakes. The combination of feedback with machine-learning proved successful. A later training session produced the rule given below.

LS

m	t	b	q	p
*	long	*	*	*

RSOPS

m	t	b	q	p
*	*	*	*	assgn

RSOPRNDS

m	t	b	q	p
*	*	*	*	2

Since by this time the interactive process had been identified as long by the bf rules, it was correctly assigned the low priority. Another batch job had entered the system by this time and was assigned the high priority because it was identified as short. The simulation ended before its behavior would have identified it as long and it would have been assigned the low priority. From this example of the results obtained for the ASC, one can say that it has attained its primary goal of adaptation to its own behavior and to its environment. The axioms and the decision matrix used by the ASC were the ones given above. All the ac rules learned have already been presented. One may

note that the ac rules do not depend completely on the axioms or the decision matrix. There is some dependence as there must be, but the ac rules also depend on the particular environment in which the ASC is executing.

The bf rules learned are the following.

bf rules

variable	operator	boundary	true	false
mode	>=	1	interactive	batch
time	>=	8	long	short
bound	>=	4	iobound	cpubound
quantum	>	1	large	small
priority	<=	5	small	large

In these bf rules, if the value from the PSV for the particular variable and the boundary value produce a true value from the operator, then the symbolic value labeled true is assigned to the SSV, else the value labeled false is assigned. These bf rules are also part of the ASC's adaptive behavior in that new bf rules can be learned and the existing rules can be modified by alterating the boundary values.

Further Example

One of the many runs of the ASC is described here as another example. This is different from the example above because another set of axioms was used. These axioms are the following.

1. increase(throughput) --> max(scheduling)

2. setpriority(high) and
   mode(interactive) --> increase(throughput)

3. setpriority(low) and
   mode(batch) --> increase(throughput)

4. setpriority(low) and
   time(long) --> increase(throughput)

5. setpriority(high) and time(short) and
   mode(interactive) --> increase(throughput)

6. setpriority(low) and
   bound(cpu) --> increase(throughput)

7. setpriority(high) and time(short) and
   bound(i/o) --> increase(throughput)

As in the above example the first job to enter the system was identified as batch, short and CPU bound. It was assigned the default priority. After the first training session the process was correctly assigned the low priority. There was no alternation of default and high priorities as with the first set of axioms. An interactive job entered the system. As in the first example, the ASC incorrectly assigned this interactive process the default priority. After two training sessions it learned a rule which captured this interactive job and correctly assigned it the high priority. A third process entered the system. It was interactive and short. It was correctly assigned the high priority. Thus the ASC again displayed adaptive behavior in learning from its mistakes and from the behavior of the system. The ac rules which it learned are the following.

LS

m	t	b	q	p
interactive	*	iobound	*	*
interactive	*	*	small	small
batch	long	*	*	*
batch	*	cpubound	*	*

RSOPS

m	t	b	q	p
*	*	*	*	assgn
*	*	*	*	assgn
*	*	*	*	assgn
*	*	*	*	assgn

RSOPRNDS

m	t	b	q	p
*	*	*	*	5
*	*	*	*	5
*	*	*	*	2
*	*	*	*	2

As one can see from this example, the axioms used have a significant influence on the behavior of the system. This system adapted to the same environment encountered by the first system, but did it more efficiently. Also, this system learned fewer ac rules. The development of effective and efficient axioms is thus an important problem for creating better adaptive schedulers.

Problems

Knowledge Base

The example above pointed out the necessity of finding good axiom sets to create efficient adaptive schedulers. One would also like the axiom sets to be general so that the scheduler would be able to handle a varied environment and be able to do so with as few axioms as possible. If the ac rules learned depend too heavily on the axioms, then the adaptive behavior of the system will be limited. One would also like to have an axiom set which could be written succinctly. This

problem of the representation knowledge base is common to all expert systems.

Overhead

Overhead is another problem. The ASC without its driving simulation contains approximately 1,000 lines of code. This is a large amount of code compared to the code which would be required to implement a conventional scheduling algorithm. However, the ASC contains code which could be used by an entire adaptive OS. For example, the ASC's procedures for creating, modifying, and matching productions could be used to learn productions for the entire OS. The major addition would be the increase in the number of axioms.

CHAPTER 6

CONCLUSIONS

Validity

Adaptability

The validity of the ASC is that it adapts to its environment and to its own behavior. Given a certain set of axioms, a decision matrix and an environment, the ASC will learn a particular set of productions to deal with this environment. With the same axioms and decision matrix, but a different environment, the ASC would create a different set of productions. For example, if only short jobs appear, the ASC will learn only productions to deal with short jobs. If, at some later time, long jobs enter the environment, the ASC learns productions for them. A nonadaptive scheduler would fail in a variable environment. Even though this is a simple example, the ASC is not limited to simple problems. It is based on Waterman's machine-learning technology and Waterman has shown that his methods can handle complex problem-solving tasks such as making the bet decision in draw poker.

Reducing Computations

As noted in Chapter 1, heuristics may be the only way to reduce the computations required by some real-time, adaptive systems. The overhead of the machine-learning technology may be more than offset by its ability to reduce computations.

Common Idea

The concept of an adaptive scheduler and an adaptive operating system is not new. Other researchers have described and created adaptive systems, so one can see that adaptation applied to systems programmming problems is an attractive and valid idea. The application of machine-learning techniques rather than numeric methods to create an adaptive system does appear to be new. It is felt that the machine-learning approach will be able to handle more complex environments than the numeric approach.

Further Research

The next logical step for the research and development of adaptive systems based on machine-learning would be to define a complex decision making problem such as the scheduler for Enslow's high-level OS which cannot be handled well by numeric methods. One would then like to show that an adaptive system similar to the one described here would be able to deal with the problem effectively and efficiently.

BIBLIOGRAPHY

BIBLIOGRAPHY

Badel, M., E. Gelenbe, J. Leroudier, and D. Potier. 1975. "Adaptive Optimization of a Time-Sharing System's Performance". Proceedings of the IEEE, 63, pp. 958-965.

Banks, J., and J. Carson, II. 1984. Discrete-Event System Simulation. Prentice-Hall, Englewood Cliffs, NJ.

Barr, A., and E. Feigenbaum. 1981. The Handbook of Artificial Intelligence, Vol I. William Kaufman, Los Altos, CA.

_____. 1982. The Handbook of Artificial Intelligence, Vol II. William Kaufman, Los Altos, CA.

Blevins, P., and C. Ramamoorthy. 1976. "Aspects of a Dynamically Adaptive Operating System". IEEE Transactions on Computers, C-25, pp. 713-725.

Bratley, P., B. Fox, and L. Schrage. 1983. A Guide to Simulation. Springer-Verlag, New York, NY.

Brinch Hansen, P. 1973. Operating System Principles. Prentice-Hall, Englewood Cliffs, NJ.

Buchanan, B., and E. Shortliffe, eds. 1984. Rule-Based Expert Systems. Addison-Wesley, Reading, MA.

Coffman, E., and R. Wood. 1966. "Interarrival Statistics for Time Sharing Systems". Communications of the ACM, 9, pp. 500-503.

Comer, D. 1984. Operating System Design, the XINU Approach. Prentice-Hall, Englewood Cliffs, NJ.

Deitel, H. M. 1983. An Introduction to Operating Systems. Addison-Wesley, Reading, MA.

Enslow, P. 1978. "What is a 'Distributed' Data Processing System?" Computer, 11, pp. 13-21.

Feigenbaum, E., and P. McCorduck. 1984. The Fifth Generation. New American Library, New York, NY.

Geck, A. 1979. "Performance Improvement by Feedback Control of the Operating System". Proceedings of the 4th Internation Symposium on Modeling and Performance Evaluation of Computer Systems. Vienna, Austria, pp. 459-471.

Gordon, G. 1978. System Simulation. Prentice-Hall, Englewood Cliffs, NJ.

Hayes, P., E. Ball, and R. Reddy. 1981. "Breaking the Man-Machine Communication Barrier". Computer, 14, pp. 19-30.

Hayes, J., and D. Michie eds. 1983. Intelligent Systems. Halsted Press, New York, NY.

Ishikawa, C., K. Sakamura, and M. Maekawa. 1982. "Dynamic Tuning of Operating Systems." Operating Systems Engineering. M. Maekawa and L. Belady eds. Springer Verlag, New York, NY. pp. 119-142.

Kaisler, S. 1983. The Design of Operating Systems for Small Computer Systems. Wiley-Interscience, New York, NY.

Kraft, A. 1984. "XCON: An Expert Configuration System at Digital Equipment Corporation." The AI Business. P. Winston, and K. Prendergast eds. MIT Press, Cambridge, MA.

Leverett, B., R. Cattell, S. Hobbs, J. Newcomer, A. Reiner, B. Schatz, and W. Wulf. 1980. "An Overview of the Production Quality Compiler-Compiler Project". Computer, 13, pp. 38-49.

Mathai, J., V. Prasad, and N. Natarajan. 1984. A Multiprocessor Operating System. Prentice-Hall International, Englewood Cliffs, NJ.

Payne, J. 1982. Introduction to Simulation. McGraw-Hill, New York, NY.

Potier, D., E. Gelenbe, and J. Lenfant. 1976. "Adaptive Allocation of Central Processing Unit Quanta". Journal of the Association for Computing Machinery, 23, pp. 97-102.

Rich, E. 1983. Artificial Intelligence. Mcgraw-Hill, New York, NY.

Ritchie, D. M., and K. Thompson. 1974. "The UNIX Time-Sharing System". Communications of the ACM, 17, pp. 365-375.

Rosin, R. 1965. "Determining a Computing Center Environment". Communications of the ACM, 8, pp. 463-468.

Sakamura, K., T. Morokuma, H. Aiso, and H. Iizuka. 1979. "Automatic Tuning of Computer Architectures". Proceedings of the AFIPS Conference, 48, pp. 499-512.

Shaw, A. 1974. The Logical Design of Operating Systems. Prentice-Hall, Englewood Cliffs, NJ.

Walter, E., and V. Wallace. 1967. "Further Analysis of a Computing Center Environment". Communications of the ACM, 10, pp. 266-272.

Waterman, D. 1970. "Generalization Learning Techniques for Automating the Learning of Heuristics". Artificial Intelligence, 1, pp. 121-170.

APPENDIX

PROGRAM CODE

```
    procedure inserttr(rn: integer; tls: leftside;
                       trsop: rsoperator;
                       trsoprnd: rsoperand; var ir: integer);
{
insert the training rule above the error-causing
rule specified by rulenumber;
}
    var
        tmpls: leftside;
        i, j: integer;
        done, match: boolean;
    begin
        if rn <= 1 then
            err(2)
        else begin
            done := false; match := false;
            i := rn;
            while not done and not match do begin
                i := i - 1;
                tmpls := ls[i];
                if tmpls[1] = blank then
                    match := true;
                if i = 1 then
                    done := true
            end;
            if not match then
                err(2)
            else begin
                ls[i] := tls; rsop[i] := trsop;
                for j := 1 to maxpsv do
                    rsoprnd[i, j] := trsoprnd[j];
                ir := i
            end
        end
    end;        { inserttr }
```

```
    procedure acop(operator: string; loprnd, roprnd: integer;
                   var modval: integer);
{
perform the operation required by
an ac rule;
}
    begin
        if operator = asssn then
            modval := roprnd
        else if operator = dec then
            modval := loprnd - roprnd
        else if operator = inc then
            modval := loprnd + roprnd
    end;        { acop }


    procedure modify(var mpsv: mpsvector; psv: psvector;
                     matchedac: integer);
{
create an actual decision,
mpsv, by loading values into
the mpsv according to the ac
rules;
}
    var
        rightsideop: rsoperator;
        i, modval: integer;
    begin
        rightsideop := rsop[matchedac];
        for i := 1 to maxpsv do begin
            if rightsideop[i] <> star then begin
                acop(rightsideop[i], psv[i], rsoprnd[matchedac, i],
                     modval);
                mpsv[i] := modval
            end else
                mpsv[i] := 0
        end
    end;        { modify }
```

```
    procedure match(ssv: ssvector; var matchedac: integer);
{
match the ssv against the ac rules;
}
    var
        left: leftside;
        i, j: integer;
        matchac, done: boolean;
    begin
        done := false; i := 0;
        while not done and (i < maxacrules) do begin
            i := i + 1;
            left := ls[i];
            if left[1] <> blank then begin
                matchac := true;
                for j := 1 to maxssv do
                    if ssv[j] <> star then
                        if (ssv[j] <> left[j])
                        and (left[j] <> star) then
                            matchac := false;
                if matchac then
                    done := true
            end
        end;
        if matchac then
            matchedac := i
        else
            matchedac := -1
    end;        { match }


    procedure getmpsv(psv: psvector; ssv: ssvector;
                      var mpsv: mpsvector; var matchedrule: integer);
{
symbolic svector is matched against all left sides
of the action rules, from top to bottom; when the first
match is made, the values of the program svector are
modified as described by the right hand side of the
matched rule;
}
    begin
        match(ssv, matchedrule);
        if matchedrule < 0 then
            err(1)
        else
            modify(mpsv, psv, matchedrule)
    end;        { getmpsv }
```

```
    function same(tls: leftside; trsop: rsoperator;
                  trsoprnd: rsoperand): boolean;
{
find out if a rule like the training rule already exits;
}
    var
        left: leftside;
        rsoprtr: rsoperator;
        i, k: integer;
        match, done, lsmatch, rsopmatch, rsoprndmatch: boolean;
        quit: boolean;
    begin
        k := 0;
        done := false; match := false;
        lsmatch := false; rsopmatch := false;
        rsoprndmatch := false;
        while not done and (k < maxacrules) do begin
            k := k + 1;
            left := ls[k];
            if left[1] <> blank then begin
                quit := false; i := 0;
                while not quit and (i < maxssv) do begin
                    i := i + 1;
                    if left[i] = tls[i] then
                        lsmatch := true
                    else begin
                        lsmatch := false;
                        quit := true
                    end
                end;
```

```
                if not quit then begin
                    rsoprtr := rsop[k];
                    i := 0;
                    while not quit and (i < maxpsv) do begin
                        i := i + 1;
                        if rsoprtr[i] = trsop[i] then
                            rsopmatch := true
                        else begin
                            rsopmatch := false;
                            quit := true
                        end
                    end
                end;
                if not quit then begin
                    i := 0;
                    while not quit and (i < maxpsv) do begin
                        i := i + 1;
                        if trsoprnd[i] = rsoprnd[k, i] then
                            rsoprndmatch := true
                        else begin
                            rsoprndmatch := false;
                            quit := true
                        end
                    end
                end
            end;
            if lsmatch and rsopmatch and rsoprndmatch then begin
                done := true;
                match := true
            end
        end;
        same := match
    end;        { same }


    procedure setdecision(psv: psvector; ssv: ssvector;
                          var mpsv: mpsvector; var acnum: integer);
{
set the decision derived from the ssv to compare with
the correct decision;
}
    begin
        setmpsv(psv, ssv, mpsv, acnum)
    end;        { setdecision }
```

```
    procedure locateec(psv: psvector; ssv: ssvector;
                       mpsv: mpsvector; var rulenumber: integer);
{
drop ssvector through all ac rules; compare
decision reached to saved associated decision;
if they are the same then that rule which produced the
decision is the error-causing rule;
if they do not match, then the error causing rule
no longer exists, so quit;
}
    var
        savedecision: mpsvector;
        i, acnumber: integer;
        match: boolean;
    begin
        for i := 1 to maxpsv do
            savedecision[i] := mpsv[i];
        getdecision(psv, ssv, mpsv, acnumber);
        match := true;
        for i := 1 to maxpsv do
            if mpsv[i] <> savedecision[i] then
                match := false;
        if match then
            rulenumber := acnumber
        else
            rulenumber := -1
    end;        { locateec }


    procedure modifybf(bfrulepntr, relvarpntr: integer;
                       psv: psvector);
{
find the relevant variable value in psvector;
replace the boundary value in the bf rule with the value
from the program subvector;
}
    begin
        if relvarpntr > 1 then begin
            if psv[relvarpntr] > bfrules[bfrulepntr].boundary then
                bfrules[bfrulepntr].boundary :=
                    bfrules[bfrulepntr].boundary + bk;
            if psv[relvarpntr] < bfrules[bfrulepntr].boundary then
                bfrules[bfrulepntr].boundary :=
                    bfrules[bfrulepntr].boundary - bk
        end
    end;        { modifybf }
```

```
    procedure createbf(psv: psvector; relevancy: rel;
                       indx, acrule: integer);
{
create a bf rule based on the psv and relevancy;
enter the number of the ac rule corresponding
to the bf rule into the rule;
}
    var
        i, emptyloc: integer;
        match, done: boolean;
    begin
        i := 0; done := false;
        match := false; emptyloc := 0;
        while not done and not match do begin
            i := i + 1;
            if bfrules[i].key = blank then begin
                emptyloc := i;
                match := true
            end;
            if i = maxbfrules then
                done := true
        end;
        if not match then
            err(3)
        else begin
            bfrules[emptyloc].key := relevancy[relvar, indx];
            if relevancy[just, indx] = small then
                bfrules[emptyloc].opr := le
            else
                bfrules[emptyloc].opr := ge;
            bfrules[emptyloc].boundary := psv[indx];
            bfrules[emptyloc].tval := relevancy[tval, indx];
            bfrules[emptyloc].fval := relevancy[fval, indx];
            bfrules[emptyloc].actionrule := acrule
        end
    end;        { createbf }
```

```
    procedure ccbfrules(psv: psvector; relevancy: rel;
                        acrule: integer);
{
create or change a bf rule; look for a bf rule
which corresponds to the relevant variable; if
such a rule exists then modify it; if no such
rule exists then create one;
}
    var
        i, j: integer;
        match, done: boolean;
    begin
        for i := 1 to maxrelvars do begin
            match := false; done := false;
            if relevancy[relvar, i] <> star then begin
                j := 0;
                while not match and not done do begin
                    j := j + 1;
                    if bfrules[j].key = relevancy[relvar, i] then
                        match := true;
                    if j = maxbfrules then
                        done := true
                end;
                if match then
                    modifybf(j, i, psv)
                else
                    createbf(psv, relevancy, i, acrule)
            end
        end
    end;        { ccbfrules }
```

```
    procedure constructrs(acceptability: accept;
                          var rsoprtr: rsoperator;
                          var rsoprnd: rsoperand);
{
create the right side (operators and operands)
of a training rule;
}
    var
        i: integer;
    begin
        for i := 1 to maxpsv do
            rsoprtr[i] := star;
        if acceptability.oprnd = prty then
            i := ppos;
        if acceptability.opr = sit then
            rsoprtr[i] := assgn;
        if (acceptability.oprnd = prty)
        and (acceptability.opr = sit)
        and (acceptability.value = high) then
            rsoprnd[i] := 5;
        if (acceptability.oprnd = prty)
        and (acceptability.opr = sit)
        and (acceptability.value = low) then
            rsoprnd[i] := 2;
    end;        { constructrs }


    procedure constructls(relevancy: rel; var ls: leftside);
{
create the left side of a training rule;
}
    var
        i: integer;
    begin
        for i := 1 to maxrelvars do
            ls[i] := relevancy[tval, i];
    end;        { constructls }
```

```
    procedure setrj(acceptability: actionstrings;
                    var relevancy: rel);
{
load the relevancy array with the relevancy
information and justification for a certain
action (acceptability); this corresponds to
the decision matrix;
}
    var
        i, j: integer;
    begin
        for i := 1 to 4 do
            for j := 1 to maxrelvars do
                relevancy[i, j] := star;
{ hypothesize that all variables are relevant; }
        relevancy[relvar, mpos] := mode;
        relevancy[relvar, tpos] := time;
        relevancy[relvar, bpos] := bound;
        relevancy[relvar, qpos] := quant;
        relevancy[relvar, ppos] := prty;
        if acceptability = setpriorityhigh then begin
            relevancy[just, mpos] := interactive;
            relevancy[tval, mpos] := interactive;
            relevancy[fval, mpos] := batch;
            relevancy[just, tpos] := small;
            relevancy[tval, tpos] := short;
            relevancy[fval, tpos] := long;
            relevancy[just, bpos] := large;
            relevancy[tval, bpos] := iobound;
            relevancy[fval, bpos] := cpubound;
            relevancy[just, qpos] := small;
            relevancy[tval, qpos] := small;
            relevancy[fval, qpos] := large;
            relevancy[just, ppos] := small;
            relevancy[tval, ppos] := small;
            relevancy[fval, ppos] := large
        end
    end;        { setrj }
```

```
    procedure constructtr(action: actionstring;
                          acceptability: accept;
                          var relevancy: rel; var tls: leftside;
                          var trsopr: rsoperator;
                          var trsoprnd: rsoperand);
{
create a training rule;
}
    begin
        setrj(action, relevancy);
        constructls(relevancy, tls);
        constructrs(acceptability, trsopr, trsoprnd)
    end;        { constructtr }


    procedure newrelsv(ir: integer; tls: leftside; ssv: ssvector;
                       psv: psvector; relevancy: rel);
{
hypothesize which variables are relevant;
}
    var
        i: integer;
    begin
        if ir > 0 then begin
            for i := 1 to maxssv do
                if ssv[i] <> tls[i] then begin
                    tls[i] := star;
                    createbf(psv, relevancy, i, ir)
                end;
            ls[ir] := tls
        end;
    end;        { newrelsv }
```

```
    procedure traininssession(action: actionstring;
                              acceptability: accept; psv: psvector;
                              ssv: ssvector; mpsv: mpsvector);
{
if the actual decision did not agree with the correct
decision then conduct a training session; this is a
simple form with no looping ( => slower conversence );
training rule is always inserted above error causing rule;
no ac rule modifications are done;
}
    var
        relevancy: rel;
        tls, left: leftside;
        trsopr, rsoprtr: rsoperator;
        trsoprnd: rsoperand;
        acrule, rulenumber, i, ir: integer;
    begin
        constructtr(action, acceptability, relevancy, tls,
                    trsopr, trsoprnd);
        ccbfrules(psv, relevancy, acrule);
        locateec(psv, ssv, mpsv, rulenumber);
        if same(tls, trsopr, trsoprnd) then
            rulenumber := 0;
        if (rulenumber > 0) and (rulenumber <= maxacrules) then
        begin
            inserttr(rulenumber, tls, trsopr, trsoprnd, ir);
            newrelsv(ir, tls, ssv, psv, relevancy)
        end
    end;        { traininssession }
```

```
    procedure transform(correctdecision: actionstring;
                        var index: integer;
                        var acceptability: accept);
{
transform the string form of the correct decision
to the record form (acceptability);
}
    begin
        if correctdecision = nop then
            index := nopval;
        if correctdecision = setpriorityhigh then begin
            index := ppos;
            acceptability.opr := sit;
            acceptability.oprnd := prty;
            acceptability.value := high
        end;
        if correctdecision = setprioritylow then begin
            index := ppos;
            acceptability.opr := sit;
            acceptability.oprnd := prty;
            acceptability.value := low
        end
    end;        { transform }
```

```
    procedure cmpdecision(mpvs: mpsvector;
                          correctdecision: actionstring;
                          var acceptability: accept;
                          var trainsignal: boolean);
{
compare the actual decision made, mpvs,
to the correct decision;
}
    var
        index: integer;
        varagree, valagree: boolean;
    begin
        varagree := false; valagree := false;
        transform(correctdecision, index, acceptability);
        case index of
            noprval:
                begin
                    varagree := true; valagree := true
                end;
            ppos:
                begin
                    if mpvs[ppos] <> nullprty then
                        varagree := true;
                    if (acceptability.value = high)
                    and (mpvs[ppos] > midpriority) then
                        valagree := true;
                    if acceptability.value = low then begin
                        if (mpvs[ppos] > minpriority)
                        and (mpvs[ppos] <= midpriority) then
                            valagree := true
                    end
                end
        end;
        if varagree and valagree then
            trainsignal := false
        else
            trainsignal := true
    end;          { cmpdecision }
```

```
    procedure matchaxs(element: actionstring;
                       var matched: boolean; var axnum: integer);
{
match axiom elements for trymaketrue;
match element against the right side of all
unused (open) axioms until have tried all or until
have found a match; as soon as find a match
make it used (closed) and return with its number; otherwise
return with 'no matches';
}
    var
        i: integer;
        done: boolean;
    begin
        matched := false;
        i := 0; done := false;
        while not done do begin
            i := i + 1;
            if (axioms[i, rs].elm = element)
            and (axioms[i, rs].mode <> 'c') then begin
                axnum := i;
                matched := true;
                axioms[i, rs].mode := 'c';
                done := true
            end;
            if i = maxax then
                done := true
        end
    end;        { matchaxs }
```

```
    procedure trymaketrue(element: actionstring;
                          var action: actionstring;
                          var quit: boolean);
{
if there was a match then
test all elements on the left side of the axiom;
if there are no more elements then quit; if the element
is an action then quit and return the action;
if the element is false; and the operator is 'and'
then quit; if the element is true; then continue;
if an element is marked unset, then call trymaketrue;
on that element;
}
    var
        axnum, i, j: integer;
        matched, done: boolean;
    begin
        j := 0;
        while (j < maxax) and not quit do begin
            j := j + 1;
            matchaxs(element, matched, axnum);
            if matched then begin
                done := false; i := 5;
                while not done do begin
                    i := i - 1;
                    if axioms[axnum, i].mode = 'a' then begin
                        done := true;
                        action := axioms[axnum, i].elm;
                        quit := true;
                    end else if axioms[axnum, i].mode = 'f' then
                    begin
                        if axioms[axnum, i].opr = 'a' then
                            done := true;
                        if i - 1 >= 1 then
                            if axioms[axnum, i - 1].opr = 'a' then
                                done := true
                    end else if axioms[axnum, i].mode = 'n' then
                        trymaketrue(axioms[axnum, i].elm, action,
                                    quit);
                    if i = 1 then
                        done := true
                end
            end
        end
    end;        { trymaketrue }
```

```
    procedure deducecorrect(var correctdecision: actionstring);
{
deduce what the correct decision should be
for a given situation;
}
    var
        quit: boolean;
    begin
        correctdecision := nop;
        quit := false;
        trymaketrue(maxscheduling, correctdecision, quit)
    end;        { deducecorrect }
```