

COMPARISON OF THE EFFECTS OF
CODING TECHNIQUES
ON SIMULATION CONCEPTS
IN PASCAL

by

BRIAN JOHN FERGUSON

B.S., University of California, Davis 1974

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree


MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1980

Approved by:


Virgil E. Wallentine
Major Professor

Spec. Coll.
LD
2668
.R4
1980
F47
c.2

TABLE OF CONTENTS

1.0	Introduction.....	1
1.1	Simulation Concepts.....	3
1.2	Concepts Implemented.....	7
2.0	Objectives.....	10
3.0	Concurrent Pascal Approach.....	11
3.1	Implementation of Simulation Concepts.....	12
3.2	Program Structure.....	15
4.0	Sequential Pascal Approach.....	20
4.1	Program Structure.....	21
4.2	Previous Approaches.....	29
5.0	Comparison of Methods.....	35
5.1	Portability.....	36
5.2	Performance.....	38
5.3	Ease of Use.....	45
6.0	Conclusion.....	50
	Bibliography.....	54
	Appendix.....	58
	Sample Concurrent Scenario.....	58
	Sequential Program Code.....	66

LIST OF FIGURES

FIGURE 1.....	6
FIGURE 2.....	18
FIGURE 3.....	19
FIGURE 4.....	22
FIGURE 5.....	25
FIGURE 6.....	30
FIGURE 7.....	33
FIGURE 8.....	39
FIGURE 9.....	42
FIGURE 10.....	43
FIGURE 11.....	46

1.0 INTRODUCTION

In this Master's Report, we will compare the effects of two coding techniques on simulation concepts in Pascal. The first technique is the use of monitors and Concurrent Pascal to implement simulation concepts. The second will use Sequential Pascal to implement the same simulation concepts using a co-routine approach.

A comparison of performance between these two techniques has been conducted. This comparison was achieved by implementing and running experiments using both techniques. The concurrent technique was previously implemented by Rich McBride at Kansas State University Computer Science Department [13]. The sequential approach was implemented by the author.

An examination has been made of other factors that affect the decision-making process in deciding between these two techniques. Examples of these factors are portability and ease of use. Each will be examined and discussed as to its impact on an intelligent choice between these two techniques.

The author's assumption will be that the reader is familiar with the Pascal language, both Sequential and Concurrent, as defined by Brinch Hansen [3],[32]. The

technical reports [10],[20] will familiarize the reader with the Pascal implemented at Kansas State University. This paper will cover those areas of simulation needed to understand the processes involved.

1.1 SIMULATION CONCEPTS

A brief overview of Simulation Concepts is necessary to understand the ideas implemented in this report. This overview will include the use and purpose of simulation, the process approach to simulation, other approaches to simulation, and a description of the facilities implemented in the simulation package described in this report.

Simulations have many uses and are used for many purposes. Simulation studies are an attempt to examine a model of the real world and real situations. The analysis of these models using simulation gives information that could not be obtained from the real world or which would be impractical to acquire. An example would be a model of aircraft performance; information which might be impossible to acquire with a real plane, or only at a high cost, could possibly be acquired with a simulation. Simulations are used for performance evaluations, as a design aid, for structural investigation, and for project planning. The accuracy of a simulation is directly related to how closely the model represents reality. This must be remembered when examining the results of any simulation.

There are many simulation languages. A general dichotomy can be made between those that are extensions of programming languages, and those that are a separate

language. The method we will examine is an extension of the Pascal Programming Language. This approach has advantages in that a programmer does not need to learn a new, distinct language, but can instead build upon his/her prior programming ability in the language to use the "new" simulation concepts.

Simulation languages are implemented in several ways. The implementation utilized in this report is the process approach. The process approach is one in which the simulation is written in terms of the flowing of an entity through various processes. An example of the process approach is described in Figure 1, the Simulation of a Car Assembly Plant.

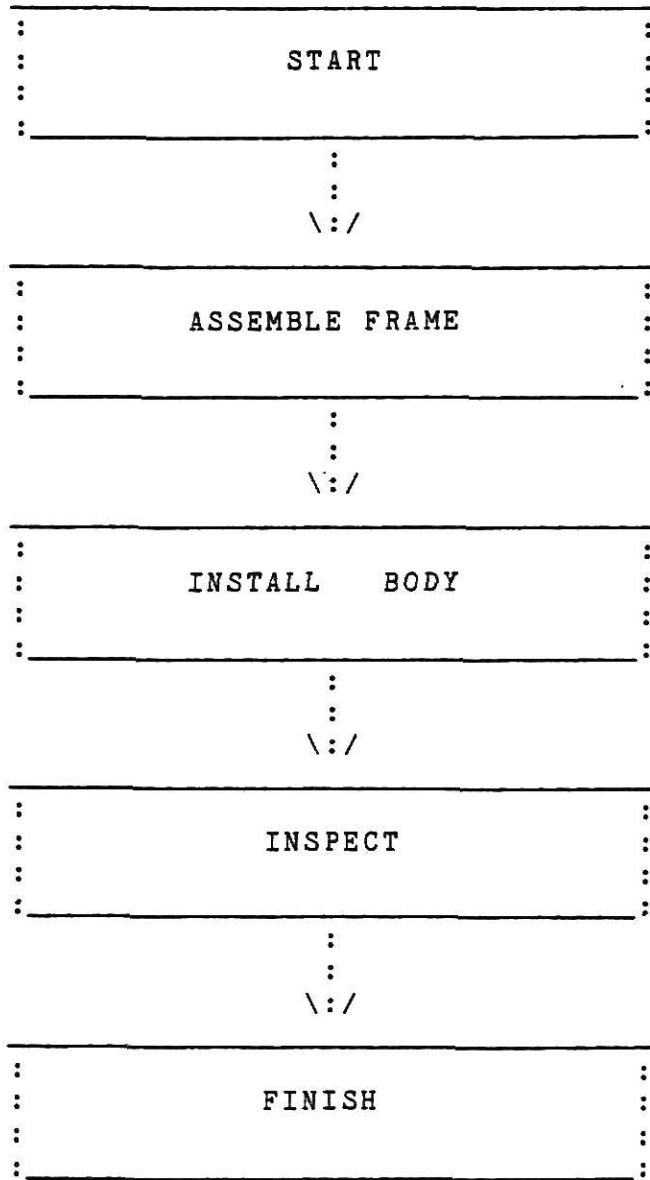
This model follows a car as it is processed or assembled. The simulation code describes the various processing steps of assembly until the entire process is finished and a completed car rolls off the assembly line. Each step is described and coded in this light. Physical resources (ex. welding machines, tröllies, etc.) have specific properties such as speed or number of actions they can do. Each process can itself be a simulation model.

The key in this approach is that accuracy is dependent upon how close the model describes reality. The process approach is the method used for the languages examined in

this report.

Other simulation methods include the Activity-Scanning Approach and the Event-Scheduling Approach [21]. Both methods are quite useful in some situations. The activity-scanning approach looks at actions to perform at a certain time or when another action occurs. The event-scheduling approach also performs actions upon the meeting of criteria for an action. Both of these methods do not have the idea of an entity that flows through the model, but rather of a series of actions to be taken when specified. In the view of activity-scanning or event-scheduling, the auto assembly plant would be a series of steps to be taken as the previous step or action is finished. This means the simulation program would be written in terms of a partial assembly: i.e., the partial assembly of the object which causes the successive step or action to occur.

FIGURE 1



SIMULATION OF CAR ASSEMBLY PLANT

1.2 CONCEPTS IMPLEMENTED

The mechanisms developed to implement simulation concepts in any language are similar. Each must implement some representation of simulation time. This internal time or clock is incremented in various fashions, but the idea is to increment the time when everything that would happen at this instant has occurred. The clock is then advanced and the actions that would occur at this instant of time are executed. This is continued until the simulation ends.

A scenario in this report refers to a section of the simulation model which deals with one aspect of the object modeled. In the example, Appendix : Sample Concurrent Scenario, the scenario describes a queue of jobs competing with other queues of jobs for CPU time and I/O time. Another mechanism that must be created is the idea of resources that are used and of limited quantity. Such resources are acquired by a simulation scenario, used for a period of time, and then returned to the resource pool and made available to any requestor. This use of resources necessitates the idea of a scenario waiting for the clock to advance before an action can take place. The resource must be controlled so that no conflicts for resources can cause deadlock within the language. Note that deadlock can be modeled, but that we cannot allow deadlock within the modeling language. This is where the ideas of seizing and

releasing resources comes from. Information about resource usage and idleness is very important to a simulation. The mechanisms that allow waiting for the clock, acquisition of a resource and release of a resource also cause information to be stored about the performance of a simulation. A more detailed description of the actual mechanisms used will be made later in this report.

A complete description of the simulation activities that have been implemented for this report can be found in the Annotated Prefix in Wallentine et. al. [13]. Some of these activities follow:

WAIT-TIME(amount)

This allows a process to wait a specified amount of simulation time before execution resumes.

SEIZE(type , number) , RELEASE(type , number)

This allows processes to engage physical resources. If a resource is in use then the process will wait until it is free.

The release does just that, it enables another process to use the resource.

WAIT_EVENT(event list) , SIGNAL(event)

These procedures allow processes to interact by allowing processes to wait until another process signals it to continue.

INC_COUNT(counter , amount) , ASSIGN_COUNT(counter , amount)

These allow processes to use counters and to keep track of the number of various actions taken.

2.0 OBJECTIVES

The objectives of this study are to conduct a performance comparison between Concurrent Pascal and Sequential Pascal simulations, to identify the other differences in the coding methods used in the report, to test the utility and usefulness of having simulation facilities added to the Pascal language, and to have available the means (i.e. programs) to test the effect of computer architecture on Concurrent Pascal and Sequential Pascal simulation programs.

3.0 CONCURRENT PASCAL APPROACH

The Concurrent Pascal program which implements the simulation facilities was written by Richard McBride, Department of Computer Science, Kansas State University. The methods used to implement the simulation concepts will be covered in this report. This discussion will be to familiarize the reader with the structures and ideas used in implementing the simulation facilities, and will be useful for both the Sequential Pascal and Concurrent Pascal approaches. A general overview of the method will be discussed; for a more detailed examination, consult the Report by Wallentine et. al. [13].

3.1 IMPLEMENTATION OF SIMULATION CONCEPTS

All simulations have several things which they must have done in order to be simulations. These functions are usually a part of the simulation language itself. This section of the report will relate the simulation concepts described in the introduction to how they are implemented in both of the approaches examined.

The first essential idea is the keeping of internal time. This internal simulation clock is manipulated by events that are waiting on the time list. The time list consists of those scenarios which have delayed themselves for a specified length of time. When an event is delayed it is placed into the linked structure of the NOTICE array. Each member of the array, when placed into or removed from the array, has several pieces of information updated or accumulated. Thus, at the end of a simulation, it can be determined how long and for what percentage of the time the event was waiting in the time list. This information plays an important part in the statistics generated by the simulation. The statistics are the main means by which judgements can be rendered about the performance of the simulation.

Another very important idea in a simulation is the need for some event to wait for another event to occur. This is

implemented with the EVENT_NOTICE variable (a record) which is placed into a data structure, and allows events to wait until an event happens or one of a list of events occurs. In either case, the record that is manipulated and placed into the event list has information concerning the length of time it waits, as well as other information which will be reported in the final statistics.

The allocation of resources is one of the major activities that a simulation must do. This is taken care of by the declaration of the characteristics of the resources available before the start of the simulation. Later when a scenario wants to use a resource it must request to use it. The SEIZE statement accomplishes this request. At the time a SEIZE is made the resource may be in use and the scenario must wait until it is free. Again the collection of information for statistical purposes necessitates much unseen bookkeeping. This allows scenarios to use resources without having to explicitly collect statistical information. The scenario may wait for a resource for a long time. Any other scenario which requests the same resource is queued up and allocated the resource on a first-come, first-served basis. When the scenario has finished its use of a resource, it is returned it to the system with the RELEASE statement. The resource is then free to be allocated to another scenario. The information about how long a scenario waits for a resource, the

utilization of a resource, and the number of scenarios waiting for a resource are maintained by the language with built-in and user transparent functions.

The methods described here apply to the implementation of both the Concurrent and Sequential approaches. This is because the methods used are easily adapted from the concurrent version to the sequential version, and because the structure of the concurrent method is very clean and understandable. The main difference between the two versions is that in the concurrent approach, the scheduling of scenarios is done in part by the concurrent mechanisms involved in the monitor concept. In the sequential approach this must be done explicitly in the actual program.

3.2 PROGRAM STRUCTURE

The Concurrent Pascal program consists of a monitor which interfaces with the file system of the host operating system, a simulation monitor which contains all of the simulation procedures and internal variables of the simulation, and an array of job processes which take a Sequential Pascal program and execute it. The simulation monitor contains all of the actual simulation procedures. The data structures needed to keep track of each process and other internal variables, such as internal time, are the monitor local variables. Each simulation scenario, which is in a job process with a simulation prefix, can call the entry procedures of the simulation monitor. The simulation monitor and file monitor both access the user's console for the typing of various messages. All printed output for the job processes and scenarios are processed through a portion of the simulation monitor. A diagram of the system components and their interaction is contained in Figure 2.

The job processes are an array of identical code. Each process loads a sequential program from the file system and then executes it. The first job process is the master process which specifies and declares counters, facilities, and ending time. All the other processes are user scenarios with each job process containing one scenario. The scenario is a user written Sequential Pascal program. Any Sequential

Pascal program may be used as a scenario. The change to make the program into a simulation program is to include calls to the entry procedures of the simulation monitor. This is accomplished with a prefix added to each program. (See Figure 3).

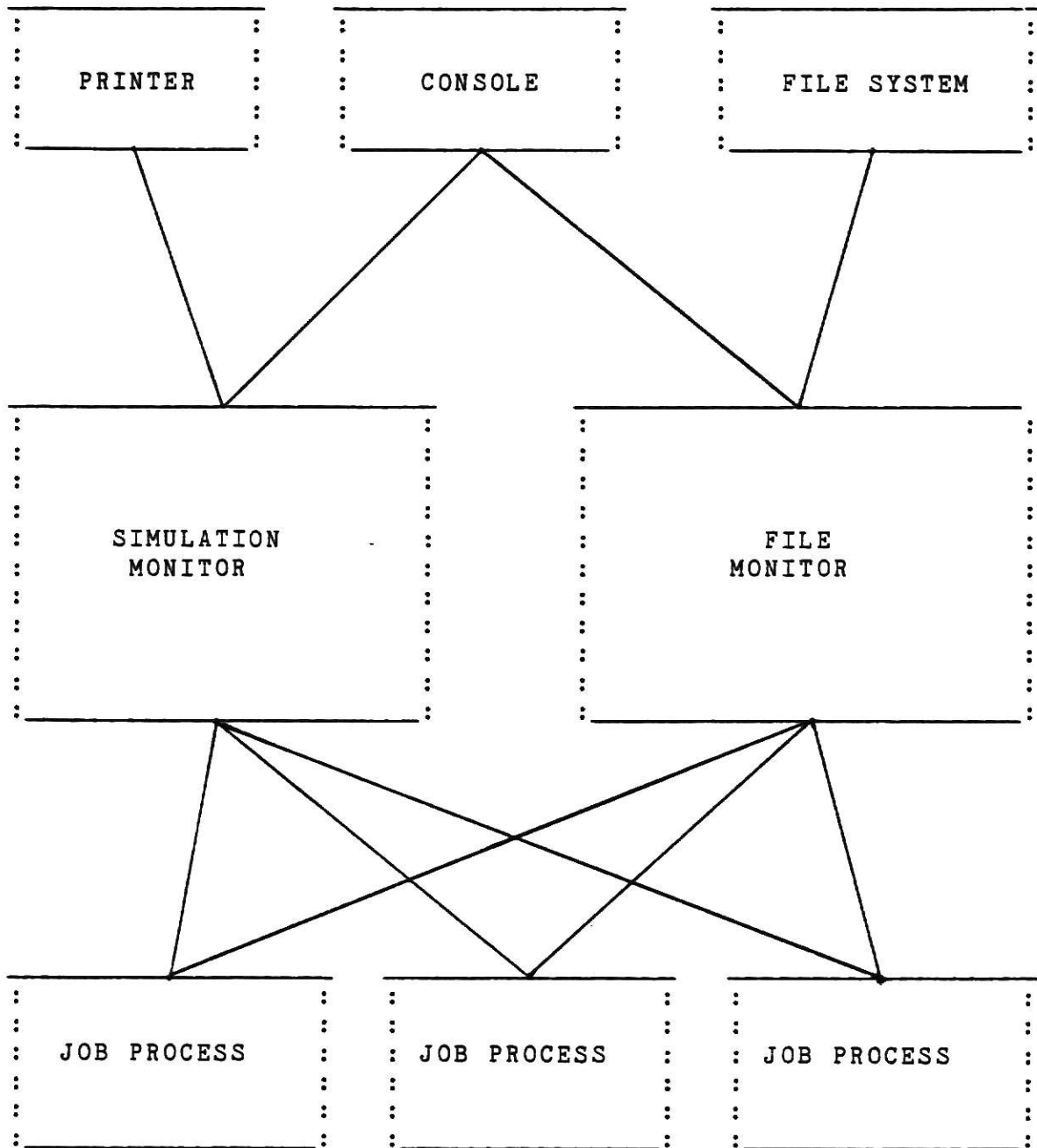
The coding technique and usage is quite similar to other languages. This simulation language consists of additions to the standard Sequential Pascal language. A programmer familiar with Sequential Pascal should have little difficulty in acquiring an understanding of the "new" statements needed. These statements are interspersed in the code, and except for these new statements (calls to the simulation monitor) and the prefix, the code is identical to normal Sequential Pascal. These Sequential Pascal programs are then inserted into the Concurrent Pascal version and executed. A sample scenario from the Concurrent Pascal approach is included in Appendix: Sample Concurrent Scenario.

The approach needed to understand the model and to code the model is straightforward. In the example of the car assembly plant, the model revolves around a car that is processed through the assembly plant. A scenario in the assembly plant would be a trolley that begins by having a car frame assembled on it. This trolley would then request resources (workers, welding, etc.) and wait at specific

stages in the assembly for actions to occur. The aspect of engine assembly could be another scenario and the car trolly would wait until the engine scenario signaled that an engine was available to be assembled to the car. This ability to tie other processes into the simulation with minimal effort makes this one of the easier simulation languages to use and expand.

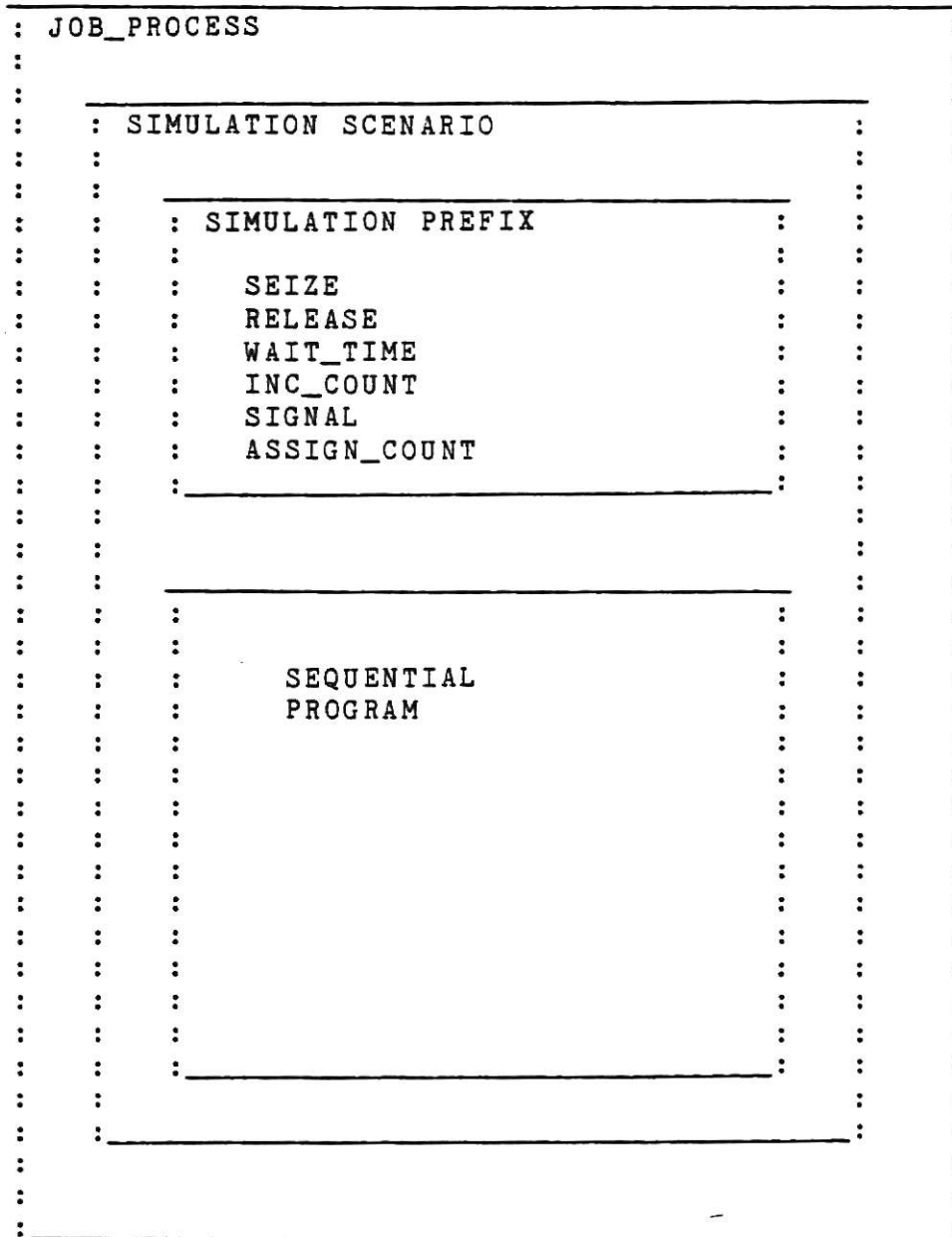
For futher information and a complete description of the simulation system consult Wallentine et. al. [13]

FIGURE 2



CONCURRENT COMPONENTS

FIGURE 3



CONCURRENT JOB PROCESS

4.0 SEQUENTIAL PASCAL APPROACH

A Sequential Pascal program which implements the same simulation features described in Section 1.1 Simulation Concepts was written by the author. The greater portion of the code is a modification of the code written by McBride [13]. The major changes were in implementing the scheduling and scenario interactions portions. The techniques for using this simulation language are quite different from the Concurrent Pascal approach and will be discussed in detail. The physical makeup of the program will also be discussed. The approaches to the problem that did not work will also be described.

4.1 PROGRAM STRUCTURE

The general organization of the Sequential Pascal program is divided into four general pieces (See Figure 4). The first part consists of the simulation procedures, which are almost identical to the Concurrent Pascal simulation monitor procedures. The second portion of the Sequential Pascal code is the scenarios that will be executed. Their purpose is the same as the scenarios in the Concurrent Pascal program. They will be discussed, in that there are several unique ideas involved in their construction. The third and fourth areas of the Sequential Pascal approach are inside a repeat statement which continues until the simulation is cancelled. These last two sections comprise the Procedure GO. The third portion is the code which calls the scenario to be executed and then calls the appropriate simulation procedure in the first section. This is implemented as a large case statement. The fourth and last section contains the scheduling procedure. This is an interesting area because of the changes required. The Concurrent Pascal approach relied on the implicit scheduling done by the language itself; however the Sequential Pascal program must implement this scheduling explicitly.

FIGURE 4

```
:  
: SIMULATION  
: CODE  
:
```

```
:  
: SCENARIO  
: CODE  
:
```

```
: PROCEDURE GO  
: REPEAT  
:  
: : CASE  
: : SCENARIO  
: : CALLS  
: : CASE  
: : SIMULATION  
: : CALLS  
:  
:  
:  
: :  
: : SCHEDULING CODE  
: :  
: :  
: UNTIL CANCELLED  
:
```

SEQUENTIAL COMPONENTS

The simulation code has within it all of the procedures of Brinch Hansen's Class Fifo. These were needed to allow the procedures in the code to use a fifo queue to keep track of records internally. The procedures from the Concurrent Pascal simulation monitor were used with very few modifications. The only changes were that Concurrent Pascal's process control (DELAY, CONTINUE, and EXCHANGE) statements had to be replaced with the Sequential Pascal scheduling used in this program. Some of these alterations resulted in the splitting of procedures, so as to comply with the Sequential Pascal language constraints. The use of the procedures in this section is identical in function to the Concurrent Pascal version's usage. Scenarios call these procedures to accomplish some simulation activity. The difference is that such calls are not made directly but rather, are made through another level of software.

The second area is the scenarios themselves. The Concurrent Pascal version loaded the scenarios from the file system. In the Sequential Pascal Language, the scenarios must be internal to the program as the language does not allow for external code to be inserted into a program at run time. The method of writing scenarios and, the scenarios use of the simulation facilities is quite different from normal coding methods.

Each scenario has the overall structure of a case

statement. Each scenario is called by the scheduling code when it is to execute. The scenario executes until it must use a simulation facility, at which time it places the parameters for such an action into the SIM_INSTR_REC variable, places the code for the action desired into the SIM_MON_INSTR variable, and then increments its INSTR_PTR variable. The scenario returns control to the scheduling code which in turn calls the appropriate simulation facility. When a scenario is recalled, it resumes execution in the next case of the case structure (See Figure 5). Control flow within the scenarios is complicated by this arrangement. The programmer, in effect, controls execution by manipulating the INSTR_PTR variable. The IF-THEN-ELSE statement becomes a change in the INSTR_PTR variable when the flow of control in a scenario was to be modified by the statement. This closely resembles the use of low-level language constructs.

Another difference, and possible problem, in the scenario is caused by the possibility of multiple copies of a scenario. This is corrected by requiring that there are no local variables in a scenario. They must be defined globally in an array. This allows multiple copies of a scenario to exist by increasing the size of the variable array thus, each call to a scenario requires an index to be passed to the scenario so that the correct element of the array of variables is used.

FIGURE 5

```
CASE INSTR_PTR [ XX , XX ] OF

  1:  BEGIN;

        SIM_INSTR_REC      :   = XXXXXX ;
        SIM_MON_INSTR      :   = XXXXXX ;
        INCR ( INSTR_PTR [ XX , XX ] ) ;
    END;

  2:  BEGIN;

        SIM_INSTR_REC      := XXXXXX ;
        SIM_MON_INSTR      := XXXXXX ;
        INCR ( INSTR_PTR [ XX , XX ] ) ;
    END;

  3:  BEGIN;

        SIM_INSTR_REC      := XXXXXX ;
        SIM_MON_INSTR      := XXXXXX ;
        INCR ( INSTR_PTR [ XX , XX ] ) ;
    END;

  4:  BEGIN;

        SIM_INSTR_REC      := XXXXXX ;
        SIM_MON_INSTR      := XXXXXX ;
        INCR ( INSTR_PTR [ XX , XX ] ) ;
    END;

  5:  BEGIN;

        SIM_INSTR_REC      := XXXXXX ;
        SIM_MON_INSTR      := XXXXXX ;
        INCR ( INSTR_PTR [ XX , XX ] ) ;
    END;
END "CASE";
```

SEQUENTIAL SCENARIO CONSTRUCTION

The third section of the Sequential Pascal version is in the procedure GO (See Figure 4). It consists of two case statements inside a repeat statement. The first case statement calls the active scenario, which is recorded in the SCENARIO_NUM variable. Upon return from the scenario the second case statement is executed. This case statement calls the appropriate simulation procedure, described in the first section. It executes this call with the appropriate parameters placed in the PARAM variable by the scenario. Upon return from the simulation procedure, control passes to the scheduling portion of the code.

The scheduling code, the fourth section, is the last half of the procedure GO. It is contained in the same repeat statement as the third section. A brief description of how a scenario would delay or continue is needed to understand the scheduling algorithm.

If a scenario delays itself, it places the current value of the SCENARIO_NUM variable into a NOTICE variable and correctly links the NOTICE into the data structure used (there are two possible data structures, the time list and event list). The scenario then sets the boolean variable DELAY to true and returns to the third section of the code.

If a scenario is to CONTINUE another scenario, it places its own number into the ACTIVE_PROCESSES fifo queue.

It then replaces the scenario number in the SCENARIO_NUM variable with the number of the continued scenario, after delinking the data structure correctly. Control returns to the third section and it is the continued process which is called next.

The EXCHANGE statement is implemented by performing the actions of a DELAY as detailed above, except that the DELAY variable is not changed. A CONTINUE is then done on the other scenario.

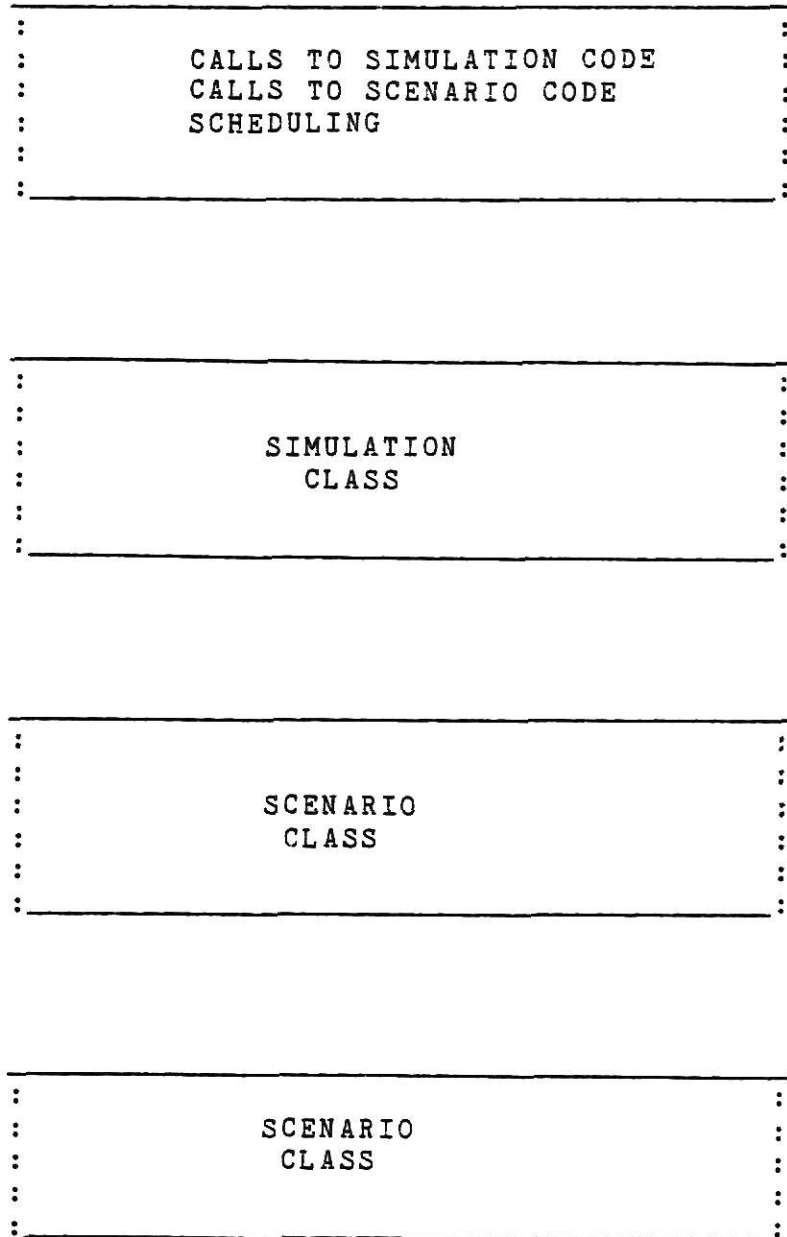
The code for the scheduling works in a straight forward manner. If a Delay has occurred, then scheduling is done. If there was no delay, then control is given back to the first case statement and the scenario is recalled. If the current scenario has delayed itself, then the ACTIVE_PROCESSES queue is checked for an active scenario number. If there is an active scenario in the queue, then the scenario number is placed into the variable SCENARIO_NUM and control is returned to the third section. If no scenarios are active, then the next scenario on the time list must be found. The procedure TIME_MOVES_ON does this. The scenario number is found in the time notice. The current simulated time is updated by the procedure UPDATE_TIME. The correct scenario is then called by the first case statement in the third section.

The techniques used in programming in this approach are somewhat unique. The Pascal language is usually considered a structured language with high level programming very much in evidence. This is true of this implementation; however, there are also areas where this has been intentionally modified to low-level programming structures. Of particular interest is the method of using the INSTR_PTR variable in the scenarios to keep track of the point of execution. This is a step backwards almost to the assembler-level of programming. In this implementation, I found it very easy to initially write the scenarios as sequential programs and to call the simulation procedures needed with the name of the function wanted. I then inserted the case form, instruction pointer incrementation, and setting the correct values into the parameter passed. This re-writing could be handled in part by a pre-processor, but that was not examined by this report.

4.2 PREVIOUS APPROACHES

There were other earlier attempts by the author to implement this approach. They were not successful, yet they did contribute knowledge that helped to find an implementation that worked. The first few attempts were to try to shift the Concurrent Pascal code to the Sequential Pascal code with no real change in the program. The problems in the handling of process control (DELAYS and CONTINUEs) very quickly resulted in these methods being abandoned. The major attempt that seemed to work was to use a Kansas State University Department of Computer Science implementation of a CLASS in Sequential Pascal to encapsulate the data of the scenarios. The approach was to have a small sequential program with several classes that would contain the scenario code and variables, as well as one which would contain the simulation code from the Concurrent Pascal simulation monitor (See Figure 6).

FIGURE 6



ORIGINAL APPROACH

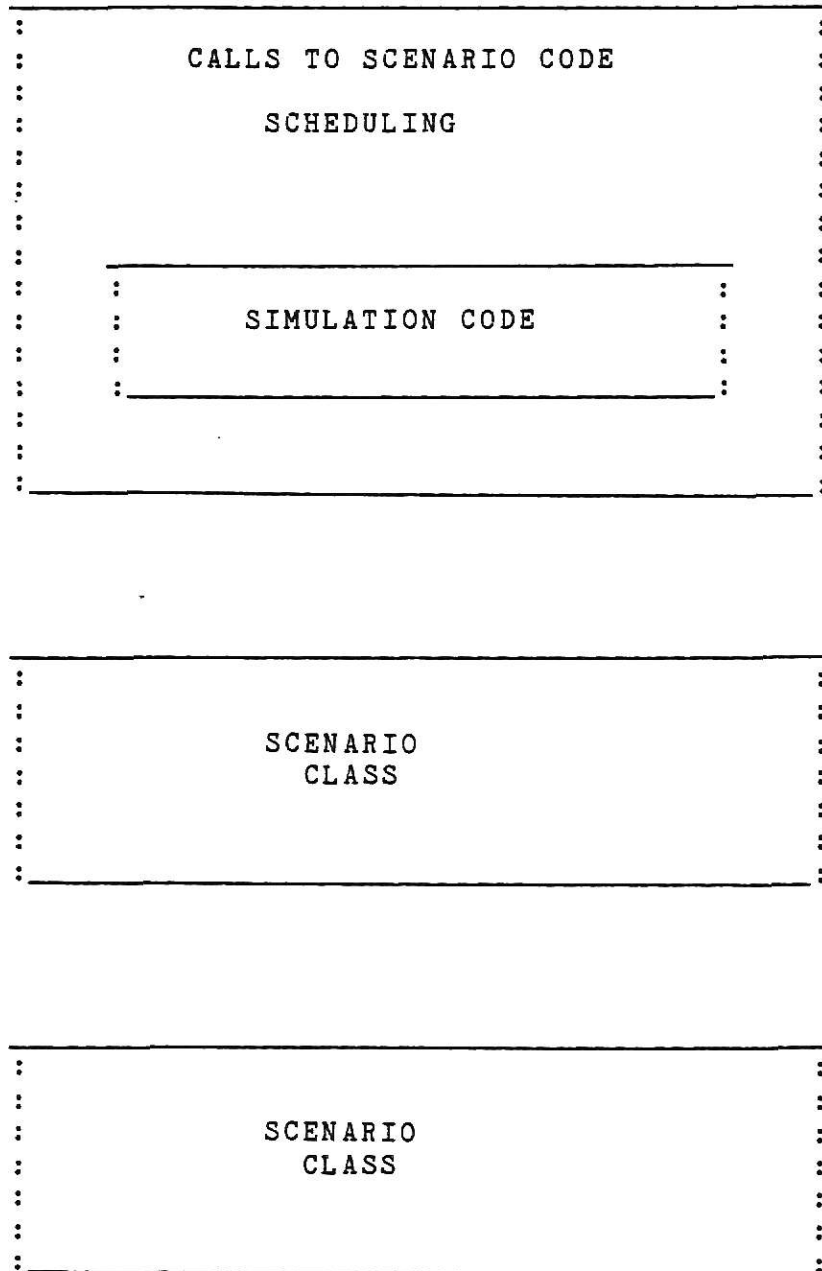
This approach made sense in that the class would allow the encapsulation of the local variables and data of the scenarios and simulation. The problem which was to prove too difficult to handle was the need for certain information in the simulation class to be available to the scenario classes while at the same time, for scenario information to be available to the simulation class in order to correctly schedule scenarios. A solution attempted was to remove the simulation class and to just encapsule the scenarios (See Figure 7). This solved the problem of scheduling information.

The next problem discovered was involved with the implementation of the CLASS concept. The CLASS local data space was removed from the memory when not in use and local variable contents would be lost. On entry, the local variables would be reset to initial values and nothing would be accomplished. The solution to this problem was to declare the variables as global to the CLASS.

Another problem with the CLASS concept was that the internal structure of the class would create problems in allowing execution to continue after a return to the main routine. The solution picked was to construct a case structure in the scenario code so that execution could continue after a return by using the case variable to point to where execution should resume. The case variable would

need to be global to the CLASS to allow a value to be saved from call to call.

FIGURE 7



TRIAL SOLUTION

The end result of all these problems, and their solutions, was that all the advantages of using the CLASS concept were removed. Everything except the code has been removed to the global procedure and is thus unprotected. The last change was to drop the CLASS idea altogether for greater portability, as the CLASS concept is not widely known or recognized in Sequential Pascal.

A partial solution for the protection problem was identified. This was more of a coding technique than a formal protection. All variables in a scenario are prefaced with a S_. This allows scenarios to be quickly checked for using only local variables. Local procedures are also named the same way. This convention could possibly be checked with some kind of preprocessor, but this was not dealt with by this report.

5.0 COMPARISON OF METHODS

The two methods used in this report have been presented. Each method and its physical structure has been explained. In this section of the report, we will compare the two methods. There are many facets of each approach that are similar and many that are different. We will examine these facets by dividing them generally into three areas (portability, performance, and ease of use), and discuss each in detail. Portability is the idea of being able to move these facilities to other computers with little or no modification, and the idea of compatibility with other Pascal languages. Performance will deal with the comparison of the actual performance between the Concurrent Pascal and Sequential Pascal versions. Ease of Use includes other facets of the program structure. Understandability, clarity of concepts, and software engineering practices are examples of the topics discussed in the Ease of Use section. The result of these discussions should be an understanding of the differences and trade-offs involved between these two approaches.

5.1 PORTABILITY

Portability is the ability to transfer a program from one computer to another with little modification. The idea of portability is one of the basic reasons for the standardization of computer languages. The programs developed and used to implement this report contain nothing which is a strictly local variation of Pascal. It should be noted that there is, at present, no standard Pascal, though there has been some movement toward standardizing Pascal [30]. The literature available indicates that the implementation of Pascal at Kansas State University contains nearly all of the features of the standard being considered. The language implemented here also contains some features not in the proposed standards. Both programs contain only the tentative standard Pascal, which should allow future porting of these simulation techniques.

The Sequential Pascal version contains only one machine dependent activity, the writing of the output. The Sequential Pascal version outputs an operating-system call to initiate output. This call is the instruction to the operating system to perform an output operation. This code is very isolated and would be quite easy to modify for a different operating system. The proposed standard Pascal would eliminate this code as the standard includes output mechanisms.

The Concurrent Pascal version has the same mechanism for output, which is well isolated from the rest of the program. The Concurrent Pascal version uses the implementation of Brinch Hansen's [32] Class, Monitor, and Process. These constructs are well known but may involve some problems in porting to other Pascal compilers. The Concurrent Pascal version also reads the scenario's code into the JOB_PROCESS from the Operating System File Subsystem. This code is isolated in the FILEMON Monitor and should be relatively easy to modify.

The points listed above are the probable problems in attempting to port this simulation code. The report by Neal and Wallentine [2] gives some more points to consider in the porting of Concurrent Pascal programs from one machine to another. These problems are not great and should be easily handled by a competent systems programmer, since they involve the communication with the operating system. I feel the portability of either system is a very strong point in favor of its greater use and acceptance. The problems involved in the porting of the Concurrent Pascal version stem mainly from the fact that there is currently no standard for Concurrent Pascal. Work has been done for a standard for Sequential Pascal but little has been done on Concurrent Pascal.

5.2 PERFORMANCE

The performance of the two versions was tested on an Interdata 8/32 computer, and was measured by implementing identical problems using both methods. Each method was executed and the time necessary to execute the program was recorded. Each program was run three times and the average time was used for the comparisons (See figure 8). It should be noted that this average was very simple to calculate as in all cases two of the run times were identical with the third time differing by a maximum of one second.

The test programs simulated a computer operating system. The models were relatively simple and were identical to the model example in Wallentine et. al. [13]. The simulations were run for various lengths of time, those chosen being 10,000, 20,000, 40,000, and 80,000 units of time. The programs were only run when there were no other programs executing on the computer. This measure of performance is crude but does allow valid comparisons to be made between the two approaches. A better measure of efficiency could be made by the insertion of a measurement package on the computer system used. At present there is no operational measurement package on the computer used in this report at Kansas State University.

FIGURE 8

COMPARISON OF RUN TIMES WITH NORMAL KERNEL

TIME UNITS	SEQUENTIAL SECONDS	CONCURRENT SECONDS	% DIFFERENCE IN PERFORMANCE
10000	48	83	73
20000	93	160	72
40000	185	314	72
80000	367	618	68

AVERAGE RUN TIMES

The performance of the two systems relative to each other shows marked differences, the sequential code running considerably faster than the concurrent code. There are several possible reasons for this difference. The Concurrent Pascal version is implemented using a program called KERNEL to interpret the concurrent operations into code suitable for a register machine. The Interdata 8/32 computer, on which both versions were tested, has a register architecture. The Sequential Pascal version does not use the same KERNEL program. The Sequential Pascal compiler generates code suitable for a register machine. The KERNEL for the Sequential Pascal version is a run time library for system support. The concurrent KERNEL is an actual program that records necessary information in order to implement the concurrent concepts involved in the language. The concurrent KERNEL also contains the run time library, but the routines differ from the sequential routines.

The approximate 70% increase in run time seemed to be excessive. A second test was undertaken using a different KERNEL program. This KERNEL program had removed from it certain features (time-slicing) that normally were present. The removal of these features in no way impaired the language as these features were user transparent and were used as system support. The results of the tests run with these changes are listed in Figure 9. The removal of the features from the KERNEL did improve the run times but

not as much as was expected.

A simpler KERNEL, such as the kernel of the semantics of MODULA, is expected to improve performance in the Concurrent Pascal version [33]. This is a large effort and is therefore the subject of a separate report.

Since the results of the comparison of the run times with the altered KERNEL also seemed to be too high, it was decided to test the Concurrent Pascal version again. This test was to code the entire program using Concurrent Pascal and not to use the Sequential Pascal programs in the scenarios. The scenarios were physically coded into a Process and not read in as were the normal concurrent version scenarios. This approach allows the slowing effects of prefix calls, file access, and loading times to be removed. It also does away with the ability to modify the simulation by simply changing scenarios. The entire code, including the simulation monitor, must be recompiled with each run. The tests were conducted with the same method used for the previous tests. The effects of the two different KERNEL programs were also checked. The results were tabulated (See Figure 10) with the times recorded as the average of three separate runs at each simulation time length. Each of these times is the result of two times being identical with the third time being within one second.

FIGURE 9

COMPARISON OF RUN TIMES WITH MODIFIED KERNEL

TIME UNITS	SEQUENTIAL SECONDS	CONCURRENT SECONDS	% DIFFERENCE IN PERFORMANCE
10000	48	77	60
20000	93	152	63
40000	183	300	64
80000	367	595	62

AVERAGE RUN TIMES

FIGURE 10

TIME UNITS	NORMAL KERNEL		% DIFFERENCE IN PERFORMANCE
	SEQUENTIAL SECONDS	CONCURRENT SECONDS	
10000	48	55	15
20000	93	105	13
40000	183	203	11
80000	367	402	10

TIME UNITS	MODIFIED KERNEL		% DIFFERENCE IN PERFORMANCE
	SEQUENTIAL SECONDS	CONCURRENT SECONDS	
10000	48	52	8
20000	93	100	8
40000	183	196	7
80000	367	388	6

COMPARISON OF AVERAGE RUN TIMES WITH BOTH KERNELS

This improvement of the concurrent running times is very significant. The modification of coding the entire simulation in Concurrent Pascal results in very efficient code. Part of this efficiency is from the removal of the Prefix code in the JOB_PROCESS code. This efficiency however displaces some other important factors. There is some loss of protection with the removal of the Prefix, and the loss of rapid and easy modification. These factors indicate that the ease of using the hybrid method, concurrent version using sequential programs as scenarios, is a major point in its favor.

5.3 EASE OF USE

This section of the report will deal with other areas of comparison between the two approaches, specifically understandability, expertise needed, clarity of concepts, and software engineering practices. Each of these areas will contribute to a greater understanding of which approach is better.

The understanding and clarity of concept are related to the fact that Pascal is the implementation language. The Concurrent Pascal Version embodies simulation entities in concurrent programming concepts. This results in a program that is very easy to understand from a concurrent point of view. The same level of understanding takes a little longer in the Sequential Pascal version. Figure 11 contains the code for a concurrent scenario with the equivalent code for a sequential scenario.

This understandability issue results from the problems in implementing co-routines in a language that does not support concurrency. The Concurrent Pascal code, with its use of Sequential Pascal programs as scenarios in the job processes, follows very closely the methods of other simulation languages in coding a model [21].

FIGURE 11

```
INC_COUNT(1,1);
SEIZE(DISK,JOB_INDEX);
WAIT_TIME(IO_TIME(.JOB_STEP.));
RELEASE(DISK,JOB_INDEX);
INC_COUNT(1,-1);
```

A. SAMPLE CODE FROM A CONCURRENT SCENARIO

```
12: BEGIN
    SIM_MON_INSTR := 16;
    E := 1; F := 1;
    "INC_COUNT"
    INCR ( N );
END;
13: BEGIN
    SIM_MON_INSTR := 9;
    E := ENUM_TO_INTEGER ( S_DISK);
    F := S_JOB1_INDEX ;
    "SEIZE"
    INCR ( N );
END;
14: BEGIN
    SIM_MON_INSTR := 5;
    E := S_JOB1_IO_TIME [ INSTANCE ,
        S_JOB1_JOB_STEP [ INSTANCE ] ];
    "WAIT_TIME"
    INCR ( N );
END;
15: BEGIN
    SIM_MON_INSTR := 10;
    E := ENUM_TO_INTEGER ( S_DISK );
    F := S_JOB1_INDEX ;
    "RELEASE"
    INCR ( N );
END;
16: BEGIN
    SIM_MON_INSTR := 16;
    E := 1; F := -1;
    "INC_COUNT"
    INCR ( N );
END;
```

B. EQUIVALENT CODE FROM A SEQUENTIAL SCENARIO

COMPARISON OF CONCURRENT AND SEQUENTIAL SCENARIO CODE

The approach used for the sequential code is not as understandable. This leads to the estimate that the Concurrent Pascal version is easier to understand and seems simpler to the user.

Earlier in this report, the idea was mentioned that the Concurrent Pascal version was easier to use, resulting from the similarities of this to other simulation languages. A programmer familiar with Concurrent Pascal should have little difficulty in learning the additions to Concurrent Pascal to use the concurrent version. This ease directly relates the approach to the simulation from the users viewpoint. The Concurrent Pascal approach is also cleaner and simpler than the Sequential Pascal approach.

The methods used to develop the simulation model from the real situations are the same for both approaches. Each required that a limited resource be allocated and used over a period of time, and that the competition for those resources is the focus of the real activity being modeled. This method for the Concurrent Pascal and Sequential Pascal versions results in Sequential Pascal programs which describe a portion of the real activity. The concurrent version scenarios are written with the actual procedure calls to the simulation code, while the sequential version scenarios are written using a cumbersome process of setting an instruction pointer, a parameter for which simulation

code is to be executed, and a record with the information to be passed to the simulation code. The concurrent version utilizes standard control statements. The sequential version scenarios must directly manipulate the instruction pointer variable in order to effect the flow of control in the scenario.

The programs are written in Pascal, which is normally considered a very structured language. It is also considered a good language in the sense of software engineering practices. The modularization of the code to prevent unwanted or unintentional interaction is one idea of software engineering. The concurrent version code fits this concept very well. The scenarios are inside job processes and are isolated from the simulation code. One idea behind modularization is the monitor concept of protecting the monitor's variables and code space.

The isolation between the parts of the concurrent version code provided by the Concurrent Pascal language and its enforcement by the compiler is a point in favor of the concurrent approach. The sequential version code does not have this automatic protection or compiler enforcement, resulting in the necessity that the modularization that is present be programmer-enforced.

The variables in the sequential code are almost

entirely global and only careful programming will insure that accidental manipulation does not occur. The sequential version program has a problem with the flow of control in the execution of its scenarios. The concurrent version code uses the standard control statements of Concurrent Pascal. The sequential version code must do explicit manipulation of the instruction pointer in the each scenario. This explicit control of execution is closer to the jump statement in assembler programming than to a high level language control structure. The overall result is that from the aspect of software engineering practices, the Concurrent Pascal version is much better. The sequential version code in some ways seems to be almost a step backwards. The Concurrent Pascal approach does have a disadvantage, in that Concurrent Pascal is not as widely used and accepted as Sequential Pascal.

6.6 CONCLUSION

In this report, we have examined two simulation methods and described the weaknesses and strengths of each. The general conclusion that appears is that the Concurrent Pascal approach has a better overall usefulness. The Sequential Pascal approach is more efficient, but the problems of understandability and maintainability make it the less desirable method. The sacrifice of some efficiency in order to have simplicity and generality in the implementing of a language is of value [32].

The usefulness of an extension to the Pascal language for simulation is significant. The combination of the structured nature of Pascal, with the inherent concurrency of the Concurrent Pascal language, makes this a valuable addition to the simulation discipline. The programs which implemented this report are complete and, combined with the report by Wallentine et. al. [13], will allow for the testing of these programs on other machine architectures.

The performance of the Concurrent Pascal code versus the Sequential Pascal code should improve when a computer with an architecture that is amenable to concurrent processing becomes available. The current efficiency can be attributed to the problems in implementing concurrent processes on the the architecture of the Interdata 8/32

computer. The ability to develop simulation models using Pascal extensions should make the language an even more important tool in computer programming.

Further work that could be done for a more complete examination of the extension of Pascal, to include simulation features, comprise several areas. The first would be to examine the Concurrent KERNEL program and to remove all code that is not needed. This should improve the execution times of the concurrent method.

The problems listed in the report dealing with the Sequential Pascal method could be eased with a pre-processor. This pre-processor could check that variables and procedures within the scenarios are local, and that no outside references to them are made. Another area that the pre-processor would simplify would be the construction of the case structure in the scenario. This pre-processor would accept a sequential program like a concurrent scenario and construct the instruction pointer incrementing, case labels, parameter passing, and simulation procedure calls needed, expanding a one statement call in the concurrent scenario to three or more statements in the sequential scenario. The three statements needed would be that of putting the parameters into the PARAM record, setting the SIM_MON_INSTR to the correct action, and incrementing the instruction pointer.

The major problem with the pre-processor idea deals with the flow of execution inside the scenario. This problem comes from the fact that control structures such as WHILE, REPEAT, and IF-THEN-ELSE statements which effect the flow of execution cannot be used in a sequential scenario because of the overall case statement structure. The translation of these statements may be beyond the effort that should be expended to construct such a pre-processor.

Another possible use of such a pre-processor would allow the general use of simulation in the Sequential Pascal language. This would be accomplished by having the pre-processor insert external references to the code outside the scenarios. Such a system with the rest of the code except for the scenarios as library support, would allow for the easy use of simulation in the Sequential Pascal Language.

Another area for further work would be to test the utility and performance of these two methods against other simulation languages. This comparison would give an idea as to the usefulness of this extension to Pascal. My opinion is that Pascal offers so much structured support, especially the Concurrent Pascal method, that it would be an improvement over most simulation languages. An advantage to this extension is that it is easy to add new features to the

extension that the simulator feels are needed.

A further examination that could be done is the testing of these methods on other computer architectures. Since this report dealt only with one machine, an examination of performance on other machines with differing architectures should prove to be of value. This examination could deal with the effects that architecture has on the implementation of high-level semantics.

The final area of concern is that of future performance. I feel that Pascal will continue to be a more widely accepted area of computing. The ability to add features will give an impetus to using Pascal in all areas, including simulation. I feel that the use of Pascal in simulations should increase as time goes on. When the concurrent method is tested on a machine architecture that supports concurrency, then the concurrent method should greatly out-perform the sequential method. This is because, when applicable, the concurrent processing is normally faster than sequential processing. concurrently. The current non-availability of such a machine architecture at Kansas State University was the main reason this idea was not tested.

BIBLIOGRAPHY

- [1] Kaubisch, W., Perrott R., and Hoare C., "Quasiparallel programming", Software-Practice and Experiance, Vol 6, No. 4 (July-August 1976).
- [2] Neal, D. and Wallentine, V., "Experience in Porting Concurrent PASCAL", Software Practice and Experience Vol. 8, No. 3 (May-June 1978).
- [3] Brinch Hansen, P., "The Programming Language Concurrent Pascal", IEEE Transactins of Software Engineering, Vol. 6, No. 2 (April - June 1976).
- [4] Wallentine, V. et. al., "An Overview of the MIMICS Network", Technical Report CS 77-05, Department of Computer Science, Kansas State University, Manhattan Kansas.
- [5] Knuth, D. and McNeley, J. L., "SOL--A Symbolic Language for General Purpose System Simulation", IEEE Transactions on Computers (Aug, 1964).
- [6] Gordon, G., The Application of GPSS V to Discrete Systems Simulation. Englewood Cliffs, N.J., Prentis-Hall, 1975.
- [7] Kiviat, P., Villaneuoa, R., and Markowitz, H. M., The Simscript II Programming Language, Englewood Cliffs, N.J., Prentice-Hall, 1968.
- [8] Dahl, O. J. and Nygaard, K., "SIMULA--An Algol-based Simulation Language", Communications of the Association of Computing Machinery Vol. 9, No. 9 (Sept. 1966).

- [9] OS-32 Programmers Reference Manual, Publication Number S29-613R02, Perkin Elmer Inc., Oceanport N. J.

- [10] Neal, D., North, B., and Wallentine, V., "SOLO Tutorials", Technical Report CS 77-20, Deptment of Computer Science, Kansas State University, Manhattan Kansas.

- [11] Vaucher, J. G. and Dwal, P., "A Comparison of Simulation Event List Algorithms", Communications of the Association of Computing Machinery Vol. 18, No. 4 (April, 1975).

- [12] Hankley, W., Wallentine, V., and Skidmore, A., "Distributed Network Simulation: Preliminary Model", Technical Report 79-02, January 1979, Department of Computer Science, Kansas State University, Manhattan Kansas.

- [13] Wallentine, V., Hankley, W., and McBride, R., "SIMMON - A Concurrent Pascal Based Simulation System", Technical Report 79-05, Department of Computer Science, Kansas State University, Manhattan, Kansas.

- [14] Bomball, M.R., Hallam S.F., Scriven, D.D., and Hallam, J.A., "Five Practical Guidelines for Successful Completion of Simulation Models", Data Management Vol 13, No. 8 (August 1975).

- [15] Farrel, W., McCall, C., and Russel, E.C., "Optimization Techniques for Computerized Simulation Models", Technical Report TR-1200-4-75, June 1975, CACI INC. Los Angeles California

- [16] Quincy, R., "'Here and Now' vs. 'There and Then'", Proceedings of the 1976 Summer Simulation Conference.

- [17] Crosbie, R., "Language and Program Structure in Simulation", Proceedings of the 1978 Summer Computer Simulation Conference.
- [18] Wirth, N., "On the Composition of Well Structured Programs", Computing Surveys Vol. 6, No. 4 (December 1974).
- [19] Schnieder, G., "Pascal : An Overview", Computer April 1979.
- [20] Young, R., "PASCAL/32 Language Definition", CIS Inc., Manhattan Kansas 1978.
- [21] Maryanski, F., Digital Computer Simulation. Prentice-Hall, Englewood Cliffs N.J. 1979.
- [22] Ivie, E., "The Programmers Workbench", Communications of the Association of Computing Machinery Vol. 20, No. 10 (October 1977).
- [23] Zelkowitz, M., "Perspectives on Software Engineering", Computing Reviews Vol. 10, No. 2 (June 1978).
- [24] Boehm, B., "Software Engineering", IEEE Transactions on Computers December 1976.
- [25] Ross, D., Goodenough, J., and Irvine, C., "Software Engineering : Process, Principles, and Goals", Computer May 1975.
- [26] Norlen, U., Simulation Model Building. Halsted Press, New York N.Y. 1975.

- [27] Yourdan, E., Constantine, L., Structured Design. Prentice-Hall Inc., Englewood Cliffs N.J. 1979.

- [28] Osborne, M. and Watts, R., Simulation and Modeling. University of Queensland Press, St. Lucia Queensland 1977.

- [29] Hibbard, P. and Schuman, S., Constructing Quality Software. North-Holland Publishing Company, New York N.Y. 1977.

- [30] Ravenel, B., "Toward a Pascal Standard", Computer April 1979.

- [31] Chandy, K. and Misra, J., "Distributed Simulation : A Case Study in Design and Verification of Distributed Programs", IEEE Transactions on Software Engineering Vol. SE-5, No. 5 (September 1979).

- [32] Brinch Hansen, P., The Architecture of Concurrent Programs. Prentice-Hall Inc., Englewood Cliffs N.J. 1977.

- [33] Wirth, N., "Modula : a Language for Modular Multiprogramming", Software-Practice and Experience. Vol. 7 No. 3 (April 1977).

APPENDIX : CONCURRENT SCENARIO

```
*****
* SIMPREFIX *
*****
```

"PREFACE OF CONSTANTS, TYPES, AND PRIMITIVES FOR SIMULATION PROCESSES"

"CONSTANTS"

```
CONST MAX_PROCESS = 25; "Max number of processes allowed"
      MAX_PROG = 9; "maximum number of sequential programs"
      MAX_FTYPE = 3; "maximum number of kinds of facilities"
      MAX_FACILITY = 19; "maximum number of facilities
                          allowed of any single type"
      MAX_COUNTER = 10; "maximum number of integer counters"
      MAX_EVENT = 20; "maximum number of event variables"
      MAX_TIME = 10000; "maximum integer value"
      MAX_COND = 20 "maximum conditions ever allowed";
      MAX_MACHINE = 3;
      MAX_MS_ENTRY = 32; "must be an even number"
```

"TYPE DEFINITIONS"

```
TYPE PRT_LINE = ARRAY[1..132] OF CHAR;
TYPE REAL_STR = ARRAY[1..20] OF CHAR;
TYPE INT_STR = ARRAY[1..6] OF CHAR;
TYPE IDENTIFIER = ARRAY[1..8] OF CHAR;
TYPE ARGTAG = (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTRTYPE);
ARGTYPE = RECORD
    TAG: ARGTAG;
    ARG: IDENTIFIER END"ARGTYPE";
CONST MAXARG = 10;
TYPE ARGLIST = ARRAY[1..MAXARG] OF ARGTYPE;
    PROCESS_INDEX = INTEGER;
    EVENT_INDEX = INTEGER;
    FACILITY_INDEX = INTEGER;
    COUNTER_INDEX = INTEGER;
    STIME = 0..MAX_TIME;
    FTYPE_INDEX = 0..MAX_FTYPE;
    MAX_FAC = ARRAY [FTYPE_INDEX] OF INTEGER;
    FTYPE_ENUM = (CPU, DISK);
                "enumeration of facility types"
    FNAME = (PRINTER, TAPE1, TAPE2, DISK1, DISK2,
            DISK3, DISK4, PRINTER1, PRINTER2);
    FAC_STATS = ARRAY [FTYPE_INDEX,
                      1..MAX_FACILITY, 1..4] OF REAL;
    COUNTER_STATS = ARRAY [1..MAX_COUNTER, 1..4] OF REAL;
    PROCESS_STATS = ARRAY [1..MAX_PROCESS, 1..2] OF REAL;
    PROC_VEC = ARRAY [1..MAX_PROCESS] OF BOOLEAN;
    EVENT_VEC = ARRAY [1..MAX_EVENT] OF BOOLEAN;
    CONDITION_VEC = ARRAY [0..MAX_COND] OF BOOLEAN;
    MACHINE_INDEX = 1..MAX_MACHINE;
    MS_ENTRY_INDEX = 1..MAX_MS_ENTRY;
```

APPENDIX : CONCURRENT SCENARIO

"SIM ENTRIES FOR CONTROLLING PROCESSES"

"Either ALIVE or CANCEL must be the first SIM call by each process; it initializes a status table entry."

PROCEDURE ALIVE ;

"identifies the process with a unique number"

PROCEDURE CANCEL(SIMULATION_END: BOOLEAN);

"terminates processing by failing the process"

PROCEDURE RENEW ;

"continues specified dormant process,
which was initially CANCELED"

PROCEDURE CLOSE(DEV: FNAME);

"Writes a file mark to 'DEV' and causes any subsequent
information to be written as a new file"

"SIM ENTRIES FOR SIMULATED TIME"

PROCEDURE WAIT_TIME (T:INTEGER);

"delays calling process during specified time interval
T; requires $CURRENT_TIME + T \leq MAX_TIME$ "

PROCEDURE WAIT_RAND (T1:INTEGER;T2:INTEGER);

"delays rally process for time interval T, where $T1 \leq T \leq T2$ "

FUNCTION TIME:REAL;

"returns value of current simulation time"

APPENDIX : CONCURRENT SCENARIO

"SIM ENTRIES FOR FACILITIES"

PROCEDURE DCL_FACILITIES (N:FTYPE_INDEX;
MAXI:MAX_FAC);

"N = maximum number of types of facilities to be used; MAXI declares maximum number of facilities to be used, by type; must be called before any facilities can be seized"

PROCEDURE SEIZE (NAME:FTYPE_ENUM;
I:FACILITY_INDEX);

"seizes facility I of type NAME; if facility is busy, calling process is delayed on a queue"

PROCEDURE RELEASE (NAME:FTYPE_ENUM;
I:FACILITY_INDEX);

"releases previously seized facility"

"SIM ENTRIES FOR SYNCHRONIZATION BY EXPLICIT EVENTS"

PROCEDURE WAIT_EVENT (I:EVENT_INDEX);

"calling process waits on event $I \leq N$, it is CONTINUED when next event I occurs"

PROCEDURE WAIT_VECTOR (VAR I:EVENT_INDEX;
VEC:EVENT_VEC);

"calling process waits on all events for which $VEC(I) = TRUE$ and $I \leq N$; upon return $I =$ event which first occurred"

PROCEDURE SIGNAL (I:EVENT_INDEX);

"signal event $I \leq N$ occurs"

APPENDIX : CONCURRENT SCENARIO

"SIM ENTRIES FOR SYNCHRONIZATION ON BOOLEAN CONDITIONS USING COUNTER VARIABLES;

SEE ALSO STANDARD BOOLEAN EVALUATION FUNCTION."

PROCEDURE WAIT_UNTIL (I:INTEGER);

"calling process is delayed if condition I, $0 \leq I \leq C$ is currently false; if delayed, process is continued when condition becomes true"

"NOTE: all conditions are Boolean expressions involving counter variables and evaluated by the standard Boolean evaluation function."

"SIM COUNTER VARIABLE FEATURES"

"these are used for counting simulated quantities, such as queues, and for Boolean conditions"

PROCEDURE DCL_COUNTERS (N:COUNTER_INDEX);

"declares maximum number of counters to be used; must be set before any counters are set"

PROCEDURE INC_COUNT (I:COUNTER_INDEX;
 AMOUNT:INTEGER);

"counter $I \leq N$ is incremented by AMOUNT;
NOTE: AMOUNT could be a negative quantity"

PROCEDURE ASSIGN_COUNT (I:COUNTER_INDEX;
 VALUE:INTEGER);

"counter $I \leq N$ is set to VALUE; NOTE: this erases all previous statistics for counter I"

FUNCTION TEST (I:COUNTER_INDEX):INTEGER;

"value of counter $I \leq N$ is returned"

"BOOLEAN CONDITION EVALUATION FUNCTION: the following function must be compiled into the CPASCAL program; it is used within the SIM monitor"

"STOCHASTIC FUNCTIONS PROVIDED"

FUNCTION UNIFORM(I,J:INTEGER):INTEGER;

"if $J < I$ then UNIFORM =I, otherwise it returns a random value $I \leq \text{UNIFORM} \leq J$ "

APPENDIX : CONCURRENT SCENARIO

"STATISTICS FEATURES PROVIDED:

Each entry below reports one set of statistics; For each procedure. The parameter FNAME may be either

SIM_JOURNL => output statistics to journal tape
PRINTER => output statistics to printer without
formatting

The statistics are reported in a tabular format."

PROCEDURE REPORT_FACS(NAME:FNAME);

"reports TABLE of type FAC_STATS:

TABLE [I;J;1] = % time facility I,J was held

TABLE [I;J;2] = average of length (q) of processes
queued waiting for facility I,J

TABLE [I;J;3] = maximum value of q

TABLE [I;J;4] = minimum value of q"

PROCEDURE REPORT_COUNTERS (NAME:FNAME);

"reports TABLE of type COUNTER_STATS:

TABLE (I;1) = number of times counter I was set,

TABLE (I;2) = average value of counter I,

TABLE (I;3) = maximum value of counter I,

TABLE (I;4) = minimum value of counter I"

PROCEDURE REPORT_PROCESSES (NAME:FNAME);

"reports TABLE of type PROCESS_STATS:

TABLE (I;1) = amount of simulation time process I
spent in wait_time,

TABLE (I;2) = amount of simulation time process I
spent waiting for events; facilities,
etc."

APPENDIX : CONCURRENT SCENARIO

"TRACING FEATURES"

PROCEDURE TRACE(FACILITIES,COUNTERS,EVENTS:BOOLEAN);

"inhibits or enables all tracing"

PROCEDURE TRACE_FILE (NAME:FNAME);

"specifies name of file or device to receive trace
information, may be any of PRINTR, TAPE1, DISK1, etc.;"
"This is also where all scenario output is sent."

PROCEDURE WRITE(TXT: PRT_LINE);

"write a line of text to the printer"

PROCEDURE INT_CHAR(NO: INTEGER; VAR STR: INT_STR);

"Converts a integer to characters of type INT_STR"

PROCEDURE REAL_CHAR(R: REAL; S,D: INTEGER; STR: REAL_STR);

"Converts a real number to characters of type REAL_STR"

PROCEDURE DISPLAY_TIME;

"Causes the current simulation time to be output"

PROGRAM P(VAR PARAM: ARGLIST);

APPENDIX : CONCURRENT SCENARIO

"simprefix here"

```
*****
* JOB1 EXECUTION *
*****
```

```
CONST JOB_INDEX = 1;
      MAX_JOB_STEP = 7;
      SLICE = 25;
```

```
VAR EXEC_TIME:ARRAY[1..MAX_JOB_STEP] OF INTEGER;
    IO_TIME:ARRAY[1..MAX_JOB_STEP] OF INTEGER;
    JOB_STEP, JOB_TIME, DELTA, I:INTEGER;
    INT_TXT: INT_STR;
    TEXT:PRT_LINE;
```

PROCEDURE WRITE_LINE;

VAR I:INTEGER;

BEGIN

TEXT(.131.):='(:13:)'; "CR"

TEXT(.132.):='(:10:)'; "NL"

WRITE(TEXT);

FOR I:=1 TO 130 DO

TEXT(.I.):=' ';

END;

PROCEDURE CENTER(LINE:PRT_LINE;START,COUNT:INTEGER);

VAR I,J:INTEGER;

BEGIN

FOR I:=1 TO COUNT DO

BEGIN

J:=START + (I - 1);

TEXT(.J.):=LINE(.I.);

END;

END;

BEGIN

ALIVE;

FOR I:=1 TO 130 DO TEXT(.I.):=' '; "initialize print line"

EXEC_TIME(.1.):=300; IO_TIME(.1.):=25;

EXEC_TIME(.2.):=5; IO_TIME(.2.):=25;

EXEC_TIME(.3.):=5; IO_TIME(.3.):=25;

EXEC_TIME(.4.):=5; IO_TIME(.4.):=25;

EXEC_TIME(.5.):=5; IO_TIME(.5.):=25;

EXEC_TIME(.6.):=5; IO_TIME(.6.):=25;

EXEC_TIME(.7.):=5; IO_TIME(.7.):=25;

APPENDIX : CONCURRENT SCENARIO

```

JOB_STEP:=1;
JOB_TIME:=EXEC_TIME(.JOB_STEP.);

REPEAT
  BEGIN
    INC_COUNT(2,1);
    SEIZE(CPU,1);

    CENTER(' JOB1 STARTS SLICE AND JOB_TIME IS',10,34);;
    INT_CHAR(JOB_TIME,INT_TXT);
    FOR I := 1 TO 6 DO TEXT[43+I] := INT_TXT[I];
    WRITE_LINE;

    IF JOB_TIME <= SLICE THEN DELTA:=JOB_TIME
    ELSE DELTA:=SLICE;

    WAIT_TIME(DELTA);

    JOB_TIME:=JOB_TIME - DELTA;  "calculate remaining job time"
    INC_COUNT(2,-1);

    CENTER(' JOB1 ENDS SLICE AND JOB_TIME IS',10,32);;
    INT_CHAR(JOB_TIME,INT_TXT);
    FOR I := 1 TO 6 DO TEXT[41+I] := INT_TXT[I];
    WRITE_LINE;
    RELEASE(CPU,1);
    IF JOB_TIME <= 0 "i.e., if job step is done" THEN
      BEGIN
        INC_COUNT(1,1);
        SEIZE(DISK,JOB_INDEX);

        WRITE(' JOB1 STARTS I/O (:13:)');

        WAIT_TIME(IO_TIME(.JOB_STEP.));

        WRITE(' JOB1 ENDS I/O (:13:)');

        RELEASE(DISK,JOB_INDEX);
        INC_COUNT(1,-1);

        JOB_STEP:=(JOB_STEP MOD MAX_JOB_STEP) + 1;
        JOB_TIME:=EXEC_TIME(.JOB_STEP.);
      END;
    END;
  UNTIL 1 <> 1;
END.

```


APPENDIX : SEQUENTIAL VERSION CODE

```
"%DEBUG := FALSE"
```

```
"%BUG := FALSE"
```

```
"%BREAK := FALSE"
```

```
" SVC CALL ITEMS MODIFIED FROM PATRICK IRELAND  
  FOR SIMMON BY BRIAN J FERGUSON AUG 79 KSU"
```

```
PROGRAM SEQPROJECT;
```

```
" This is the sequential code for Brian J Ferguson  
  Master's project"
```

```
(*
```

```
    This program has within it several sections of code  
    that are delimited by "%XXX" and "%END XXX".  
    These sections of code are modified by a pre-processor  
    used at Kansas State University. Conditions are defined  
    with social statements, such as the first three statements  
    of this program, as either TRUE or FALSE. If a condition  
    is TRUE then the code between a "%XX" and "%END XX" is  
    not modified and is functional. If a condition is FALSE  
    then the code between the delimiters is modified into a  
    comment. The pre-processor is very easy to use and very  
    good at enabling diagnostics and tracing statements to be  
    left in production code and available for debugging at  
    a later time. The conditions used in this program are  
    used as follows:
```

```
    DEBUG : This enables diagnostics to check on the  
            execution of the simulation.
```

```
    BUG    : This enables diagnostics on the trace of  
            execution to a more detailed degree than  
            DEBUG.
```

```
    BREAK : This enables statements that generate external  
            breakpoints for an interactive debugger.
```

```
*)
```

APPENDIX : SEQUENTIAL VERSION CODE

```

TYPE SVC1_CONTROL_BLOCK = RECORD
    FUNCTION_CODE : BYTE;
    UNIT_NUMBER   : BYTE;
    STATUS        : BYTE;
    DEVICE        : BYTE;
    START_ADDRESS : INTEGER;
    END_ADDRESS   : INTEGER;
    RANDOM_ADDR   : INTEGER;
    LENGTH_TRANS  : INTEGER;
    ITAM_USE      : INTEGER;
END;

```

```

TYPE LINTEGER = INTEGER;

```

```

CONST

```

```

    PAGELENGTH = 512;
    IDLENGTH = 8;
    READ_REQUEST = 64;
    WRITE_REQUEST = 32;
    WRITE_FILE_MARK_REQUEST = #88;
    RANDOM_ACCESS = 4;
    EXCLUSIVE_READ_ONLY = 32;
    EXCLUSIVE_WRITE_ONLY = 96;
    EXCLUSIVE_READ_WRITE = 224;
    SHARED_READ_ONLY = 0;
    ASSIGN_REQUEST = 64;
    CLOSE_REQUEST = 4;
    CR = '(:13:)';
    NL = '(:10:)';
    EM = '(:25:)';
    FF = '(:12:)';
    NULL = '(:0:)';
    BLANK = '(:20:)';

```

```

TYPE

```

```

    PAGE = ARRAY[1..512] OF CHAR;
    IDENTIFIER = ARRAY[1..IDLENGTH] OF CHAR;
    FNAME = (PRINTR, TAP1, TAP2, DISK1, DISK2, DISK3,
              DISK4, PRINTR1, PRINTR2);
    PRT_LINE = ARRAY[1..132] OF CHAR;
    MINI_STR = ARRAY[1..4] OF CHAR;
    INT_STR = ARRAY[1..6] OF CHAR;
    REAL_STR = ARRAY[1..20] OF CHAR;

```

APPENDIX : SEQUENTIAL VERSION CODE

CONST

```

MASTER_ID = 1;
SCENARIO1 = 2;
MAX_PROCESS = 10;  " = total number of scenarios
                   + master_timer + loader"
FACILITY_TYPES = 3;
MAX_FACILITY = 19;
MAX_CNTR = 10;
MAX_EVENT = 9;
PMAX_EVENT = 5;
NOTICE_MAX = 30;  ">= max_process + n_intvls + 2"
DT = 10;
N_INTVLS = 3;

```

TYPE

```

LIFE = (LIVING, DEAD);
ATTRIBUTE_ARRAY = ARRAY [ 1..MAX_PROCESS ] OF INTEGER;
MAX_FAC = ARRAY[0..MAX_FACILITY] OF INTEGER;
EVENT_VEC = ARRAY[1..MAX_EVENT] OF BOOLEAN;
EVENT_PTRS = RECORD
    PREV, NEXT: INTEGER; "back & forward pointers"
    "Position of this event in other link records"
    PREV_POS , NEXT_POS : INTEGER;
    EVENT_NO: INTEGER;
END "EVENT_PTRS";

```

```

DELAY_ELEM = RECORD
    LINK: ARRAY[1..PMAX_EVENT] OF EVENT_PTRS;
    SLEEP: INTEGER;
    WAIT_TIME, ACTIVE_TIME, REQ_TIME: REAL;
    LST_TIME: REAL;
    STATUS: LIFE ;
END "DELAY_ELEM";

```

```

FACILITY_REC = RECORD
    FREE: BOOLEAN;
    Q_HEAD: INTEGER;
    USE_TIME: REAL;
    Q: INTEGER;
    Q_AVE: REAL;
    Q_MIN, Q_MAX: INTEGER;
    SEIZE_TIME, AVE_WAIT: REAL;
    RLSE_CNT: REAL;
    USE_CNT: REAL  "= Q + number of successful SEIZES"
END "FACILITY_REC";

```

APPENDIX : SEQUENTIAL VERSION CODE

```
CNTR_REC = RECORD
  VALUE:INTEGER;
  AVE_VALUE: REAL;
  MIN_VALUE,MAX_VALUE: INTEGER;
  USE_CNT:REAL ;
END "CNTR_REC";
```

```
EVENT_REC = RECORD
  DELAYED: BOOLEAN;
  Q_HEAD: INTEGER;
  NUM_SIGNALS, NUM_WAITS: REAL;
  Q_AVE: REAL;
  Q_MAX,Q:INTEGER;
  AVE_WAIT: REAL ;
END "EVENT_REC";
```

```
NOTICE_REC = RECORD
  ID: INTEGER;
  PREV,NEXT:INTEGER;
  OCCURRENCE: REAL;
  DUMMY: BOOLEAN;
  BEDTIME: REAL;
  SLEEP: INTEGER ;
END "NOTICE_REC" ;
```

```
TYPE SIM_INSTR_REC = RECORD
  A , B , C : BOOLEAN;
  D : FNAME;
  E : INTEGER;
  F , G : INTEGER;
  H : REAL;
  I : MAX_FAC;
  J : EVENT_VEC;
  K : PRT_LINE;
  L : INT_STR;
  M : REAL_STR;
END;
```

APPENDIX : SEQUENTIAL VERSION CODE

"INSERT SCENARIO CONSTANTS AND TYPES HERE"

CONST

"JOB1TEST CONSTANTS"

S_JOB1_MAX_JOB_STEP = 7;
S_JOB1_SLICE = 25;
S_JOB1_INDEX = 1;

"JOB2TEST CONSTANTS"

S_JOB2_MAX_JOB_STEP = 1;
S_JOB2_SLICE = 25;
S_JOB2_INDEX = 2;

TYPE

"SCENARIO TYPES"

S_FTYPE_ENUM = (S_CPU , S_DISK);
"Enumeration of facility types"

APPENDIX : SEQUENTIAL VERSION CODE

VAR

```

CLOCK, OLD_TIMES : REAL;
DELAY_LST : ARRAY [ MASTER_ID..MAX_PROCESS ] OF DELAY_ELEM;
FREE_DELAYS : INTEGER;
FACILITY_INDEXER : ARRAY [ 0..FACILITY_TYPES ] OF INTEGER;
FACILITY : ARRAY [ 1..MAX_FACILITY ] OF FACILITY_REC;
CNTR : ARRAY [ 1..MAX_CNTR ] OF CNTR_REC;
EVENT : ARRAY [ 1..MAX_EVENT ] OF EVENT_REC;
NOTICE : ARRAY [ 1..NOTICE_MAX ] OF NOTICE_REC;
NFREE : INTEGER;
INTVL_PTR : ARRAY [ 0..N_INTVLS ] OF INTEGER;
ICURRENT,CURRENT : INTEGER;
LOWERBOUND : REAL;
PROCS_DELAYED, ACTIVE_PROCS : INTEGER;
NO,M : INTEGER;
CDTN_LST, CDTN_PTR : INTEGER;  "Head of condition list"
SM_TXT : MINI_STR;
LG_TXT : PRT_LINE;
INT_TXT : INT_STR;
REAL_TXT : REAL_STR;
SIM_STARTED : BOOLEAN;
STALLED : INTEGER;
TRACE_FACILITY, TRACE_EVENT, TRACE_CNTR : BOOLEAN;
CNTR_LIMIT,FCOUNT : INTEGER;
TRACE_TABLE : ARRAY [ 1..MAX_PROCESS ] OF FNAME;
SCENARIO_NUM : INTEGER;
SIM_MON_INSTR : INTEGER;
ACTIVE_PROCESSES : ARRAY [1..MAX_PROCESS ] OF INTEGER;
SPEC1 , SPEC2 , SPEC3 : ARRAY [1..MAX_PROCESS] OF INTEGER;
SPEC4 : ARRAY [ 1..MAX_PROCESS ] OF BOOLEAN;
HEAD , LENGTH , LIMIT : INTEGER;
CANCELLED : BOOLEAN;
DELAY : BOOLEAN;
INSTR_PTR : ARRAY [ 1..MAX_PROCESS ] OF INTEGER;
PARAM : SIM_INSTR_REC;

```

APPENDIX : SEQUENTIAL VERSION CODE

"INSERT SCENARIO VARIABLE DECLARATIONS HERE"

"JOB1TEST VARIABLES"

```
S_JOB1_EXEC_TIME : ARRAY [ 1..1 ] OF
                      ARRAY [ 1..S_JOB1_MAX_JOB_STEP ] OF
                      INTEGER;
S_JOB1_IO_TIME : ARRAY [ 1..1 ] OF
                      ARRAY [ 1.. S_JOB1_MAX_JOB_STEP ] OF
                      INTEGER;
S_JOB1_JOB_STEP ,
S_JOB1_JOB_TIME ,
S_JOB1_DELTA : ARRAY [ 1..1 ] OF INTEGER;
S_JOB1_INT_TXT : ARRAY [ 1..1 ] OF INT_STR;
S_JOB1_TEXT : ARRAY [ 1..1 ] OF PRT_LINE;
```

"JOB2TEST VARIABLES"

```
S_JOB2_EXEC_TIME : ARRAY [ 1..1 ] OF
                      ARRAY [ 1..S_JOB2_MAX_JOB_STEP ] OF
                      INTEGER;
S_JOB2_IO_TIME : ARRAY [ 1..1 ] OF
                      ARRAY [ 1..S_JOB2_MAX_JOB_STEP ] OF
                      INTEGER;
S_JOB2_JOB_STEP,
S_JOB2_JOB_TIME,
S_JOB2_DELTA : ARRAY [ 1..1 ] OF INTEGER;
S_JOB2_INT_TXT : ARRAY [ 1..1 ] OF INT_STR;
S_JOB2_TEXT : ARRAY [ 1..1 ] OF PRT_LINE;
```

"MASTER_TIMER VARIABLES"

```
S_MAST_FAC_SET : MAX_FAC;
```

"LOOP COUNTER FOR ALL SCENARIOS"

```
III : INTEGER;
```

APPENDIX : SEQUENTIAL VERSION CODE

PROCEDURE SVC1 (S1: SVC1_CONTROL_BLOCK); EXTERN;

FUNCTION ATTRIBUTE_GET : INTEGER;
BEGIN
 ATTRIBUTE_GET := SCENARIO_NUM;
END;

"%IF BREAK"
 PROCEDURE BREAKPNT (LINENUM : LINTEGER); EXTERN;
"%END BREAK"

"These functions are from the Class FIFO "

FUNCTION ARRIVAL : INTEGER;
BEGIN
 ARRIVAL := HEAD;
 HEAD := HEAD + 1;
 LENGTH := LENGTH + 1;
END "ARRIVAL";

FUNCTION DEPARTURE : INTEGER;
BEGIN
 DEPARTURE := HEAD - 1;
 HEAD := HEAD - 1;
 LENGTH := LENGTH - 1;
END "DEPARTURE";

FUNCTION EMPTY: BOOLEAN;
BEGIN
 EMPTY := (LENGTH = 0)
END "EMPTY";

FUNCTION FULL: BOOLEAN;
BEGIN
 FULL := (LENGTH = LIMIT);
END "FULL";

APPENDIX : SEQUENTIAL VERSION CODE

```
*****
* START OF SIMULATION CODE *
*****
```

```
PROCEDURE XSTR(String: PRT_LINE; FIRST,COUNT: INTEGER);
VAR
  I : INTEGER;
BEGIN
  FOR I := 1 TO COUNT DO LG_TXT [ PRED(FIRST+I) ] :=String [ I ] ;
END"XSTR";
```

```
PROCEDURE CHAR_STR(I: UNIV INTEGER; VAR String: MINI_STR);
VAR
  C , J : INTEGER;
BEGIN
  IF I = 256
    THEN C := I DIV 256
    ELSE C := I ;
  J := C DIV 100;
  C := C MOD 100;
  String [ 1 ] := CHR(J +48);
  J := C DIV 10;
  C := C MOD 10;
  String [ 2 ] := CHR(J+48);
  String [ 3 ] := CHR( C + 48 );
  String [ 4 ] := ' ';
END "CHAR_STR";
```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE DUMP_INT(NO : INTEGER ; VAR STR : INT_STR);
VAR
  I , J , K : INTEGER;
BEGIN
  IF NO >= 0
  THEN
    BEGIN
      STR [ 1 ] := ' ';
      I := NO;
    END
  ELSE
    BEGIN
      STR [ 1 ] := '-';
      I := ABS ( NO );
    END;
  K := 0;
  WHILE I > 0 DO
    BEGIN
      J := I MOD 10;
      I := I DIV 10;
      STR [ 6-K ] := CHR(J + 48);
      K := SUCC(K);
    END "WHILE";
    FOR I := 6-K DOWNT0 2 DO STR [ I ] := ' ';
    IF STR [ 6 ] = ' '
    THEN STR [ 6 ] := '0';
  END "DUMP_INT";

```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE WRITE_REAL(R: REAL; S,D: INTEGER; VAR STR: REAL_STR);
VAR
  I , IS , ID , OUT , DIGIT : INTEGER;
  NUMB , EXP : REAL;
  NON_0 : BOOLEAN;
BEGIN
  NON_0 := FALSE;
  OUT := 2;
  IS := S;
  ID := D;
  NUMB := ABS ( R );
  IF R < 0.0
    THEN STR [ 1 ] := '-'
    ELSE STR [ 1 ] := ' ';
  WHILE IS > 0 DO
    BEGIN
      EXP := 1.0;
      FOR I := 1 TO IS - 1 DO EXP := EXP * 10.0;
      DIGIT := TRUNC(NUMB/EXP);
      NUMB := NUMB - (CONV(DIGIT) * EXP);
      IF (DIGIT <> 0) & NOT NON_0
        THEN NON_0 := TRUE;
      IF DIGIT > 9
        THEN STR [ OUT ] := '*'
        ELSE IF NON_0
          THEN STR [ OUT ] := CHR ( DIGIT + 48)
          ELSE STR [ OUT ] := ' ';
      IS := PRED ( IS );
      OUT := SUCC ( OUT );
    END "ELIHW";
    IF NOT NON_0
      THEN STR [ PRED ( OUT ) ] := '0';
    IF ID > 0
      THEN
        BEGIN
          STR [ OUT ] := '.';
          OUT := SUCC ( OUT );
        END "FI";
    WHILE ID > 0 DO
      BEGIN
        DIGIT := TRUNC(NUMB * 10.0);
        NUMB := NUMB * 10.0 - CONV(DIGIT);
        STR [ OUT ] := CHR(DIGIT + 48);
        OUT := SUCC ( OUT );
        ID := PRED ( ID );
      END "ELIHW";
      FOR I := OUT TO 20 DO STR [ I ] := ' ';
    END "WRITE_REAL";

```

APPENDIX : SEQUENTIAL VERSION CODE

```
PROCEDURE INT_CHAR(NO: INTEGER; VAR STR: INT_STR);  
  BEGIN  
    DUMP_INT(NO,STR);  
  END "INT_CHAR";
```

```
PROCEDURE REAL_CHAR(R: REAL; S,D: INTEGER; VAR STR: REAL_STR);  
  BEGIN  
    WRITE_REAL(R,S,D,STR);  
  END "REAL_CHAR";
```

APPENDIX : SEQUENTIAL VERSION CODE

```
PROCEDURE WRITE(DEVICE: FNAME);
  " This procedure writes a 132 character line to the logical
    unit indicated by the device.  The SVC1 call to the O S was
    written by Brian J Ferguson"
```

```
VAR
  UNIT : INTEGER;
  SVC1_BLOCK : SVC1_CONTROL_BLOCK;
  TEXT_TO_WRITE : PRT_LINE;
  I : INTEGER;
BEGIN
  CASE DEVICE OF
    PRINTR : UNIT := 1;
    TAP1    : UNIT := 2;
    TAP2    : UNIT := 3;
    DISK1   : UNIT := 4;
    DISK2   : UNIT := 5;
    DISK3   : UNIT := 6;
    DISK4   : UNIT := 7;
    PRINTR1 : UNIT := 8;
    PRINTR2 : UNIT := 9;
  END "ESAC";
  LG_TXT [ 132 ] := CR;
  WITH SVC1_BLOCK DO
    BEGIN
      UNIT_NUMBER := UNIT;
      FUNCTION_CODE := WRITE_REQUEST;
      START_ADDRESS := ADDRESS (LG_TXT );
      END_ADDRESS := START_ADDRESS + 131;
    END "HTIW";
  SVC1 ( SVC1_BLOCK );
  FOR I := 1 TO 130 DO LG_TXT [ I ] := ' ';
END "WRITE";
```

```
FUNCTION TRACE_DEVICE : FNAME;
" This function is used to identify what the trace device
of a particular job process is.  "
VAR
  I : INTEGER;
BEGIN
  I := ATTRIBUTE_GET ;
  TRACE_DEVICE := TRACE_TABLE [ I ];
END;
```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE WRITE_SIM ( TXT : PRT_LINE );
  "This procedure allows the job process to write a
    132 character line
  to the device found by TRACE_DEVICE "
  VAR
    I : INTEGER;
  BEGIN
    FOR I := 1 TO 130 DO LG_TXT [ I ] := TXT [ I ];
    WRITE ( TRACE_DEVICE );
  END;

```

```

PROCEDURE WRITE_PREFACE;
  VAR
    I : INTEGER;
  BEGIN
    WRITE_REAL(CLOCK,8,0,REAL_TXT);
    XSTR(REAL_TXT,1,9);
    XSTR ( ' PROCESS ',10,9);
    I := ATTRIBUTE_GET ;
    CHAR_STR( I ,SM_TXT);
    XSTR(SM_TXT,19,4);
  END "WRITE_PREFACE";

```

```

PROCEDURE NADD(N_PTR:INTEGER);
  "Add a notice back to the free list"
  BEGIN
    NOTICE [ N_PTR ] .NEXT := NFREE;
    NFREE := N_PTR;
  END "NADD";

```

```

PROCEDURE NGET(VAR N_PTR:INTEGER);
  "Retrieve index of a free notice"
  BEGIN
    N_PTR := NFREE;
    NFREE := NOTICE [ N_PTR ] .NEXT;
  END "NGET";

```

```

PROCEDURE RAND(VAR R: REAL);
  BEGIN
    NO := 259 * NO;
    IF NO < 0
      THEN NO := NO + 32767 + 1;
    R := CONV(NO)/32767.0 ;
  END "RAND";

```

APPENDIX : SEQUENTIAL VERSION CODE

```
PROCEDURE ALIVE;  
  BEGIN  
    DELAY_LST [ ATTRIBUTE_GET ] .STATUS := LIVING;  
  END "ALIVE";
```

```
PROCEDURE RENEW;  
BEGIN END;
```

. APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE CLOSE(DEVICE: FNAME);
  " Write closed by who to file,
  if the device is the trace device write file mark"
  " added by Brian J Ferguson"
  VAR
    SVC1_BLOCK : SVC1_CONTROL_BLOCK;
    UNIT : INTEGER;
    I : MINI_STR;
  BEGIN
    I [ 1 ] := CHR( ATTRIBUTE_GET );
    XSTR(' FILE CLOSED BY ',2,16);
    XSTR(I,17,4);
    WRITE(DEVICE);
    CASE DEVICE OF
      PRINTR : UNIT := 1;
      TAP1 : UNIT := 2;
      TAP2 : UNIT := 3;
      DISK1 : UNIT := 4;
      DISK2 : UNIT := 5;
      DISK3 : UNIT := 6;
      DISK4 : UNIT := 7;
      PRINTR1 : UNIT := 8;
      PRINTR2 : UNIT := 9;
    END "CASE";
    WITH SVC1_BLOCK DO
      BEGIN
        FUNCTION_CODE := WRITE_FILE_MARK_REQUEST;
        UNIT_NUMBER := UNIT;
      END;
      IF DEVICE = TRACE_DEVICE
        THEN SVC1( SVC1_BLOCK );
    END "CLOSE";

FUNCTION UNIFORM (I , J : INTEGER) : INTEGER;
  VAR
    R : REAL;
  BEGIN
    IF J <= I
      THEN UNIFORM := I
      ELSE BEGIN
        RAND( R );
        UNIFORM := I + TRUNC(CONV(J + 1 - I) * R);
      END "FI";
    END "UNIFORM";

```


APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE TIME_MOVES_ON;
  VAR
    Y : INTEGER;
  BEGIN
    WHILE NOTICE [ CURRENT ] .DUMMY DO
      WITH NOTICE [ CURRENT ] DO
        BEGIN
          LOWERBOUND := LOWERBOUND + CONV(DT);
          ICURRENT := SUCC(ICURRENT);
          NOTICE [ PREV ] .NEXT := NEXT;
          NOTICE [ NEXT ] .PREV := PREV;
          Y := NOTICE [ INTVL_PTR [ N_INTVLS ] ] .PREV;
          OCCURRENCE := LOWERBOUND + CONV(N_INTVLS * DT);
          WHILE NOTICE [ Y ] .OCCURRENCE > OCCURRENCE DO
            Y := NOTICE [ Y ] .PREV;
          NEXT := NOTICE [ Y ] .NEXT;
          NOTICE [ NEXT ] .PREV := CURRENT;
          NOTICE [ Y ] .NEXT := CURRENT;
          CURRENT := NOTICE [ PREV ] .NEXT;
          PREV := Y;
        END "ELIHW";
      END "TIME_MOVES_ON";
    END "TIME_MOVES_ON";
  END "TIME_MOVES_ON";

```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE CANCEL(CANCEL_SIMULATION : BOOLEAN);
VAR
  I: INTEGER;
  ACTIVE_UPDATED : BOOLEAN;
BEGIN
  I := ATTRIBUTE_GET ;
  DELAY_LST [ I ] .STATUS := DEAD;
  "Write out statistics???"
  ACTIVE_PROCS := PRED(ACTIVE_PROCS);
  IF I = MASTER_ID
  THEN
    BEGIN
      FOR I := 1 TO FCOUNT DO
        WITH FACILITY [ I ] DO
          IF SEIZE_TIME > 0.0
          THEN USE_TIME :=USE_TIME + CLOCK - SEIZE_TIME;
        FOR I := SCENARIO1 TO MAX_PROCESS DO
          IF DELAY_LST [ I ] .REQ_TIME > 0.0
          THEN DELAY_LST [ I ] .WAIT_TIME :=
            DELAY_LST [ I ] .WAIT_TIME + CLOCK -
            DELAY_LST [ I ] .REQ_TIME;
        I := CURRENT;
        ACTIVE_UPDATED := FALSE;
        REPEAT
          IF NOT NOTICE [ I ] .DUMMY
          THEN WITH DELAY_LST [ NOTICE [ I ] .ID ] DO
            IF NOT ACTIVE_UPDATED
            THEN
              BEGIN
                ACTIVE_UPDATED := TRUE;
                ACTIVE_TIME := ACTIVE_TIME
                  + CLOCK - OLD_TIMES;
                LST_TIME := LST_TIME + OLD_TIMES -
                  NOTICE [ I ] .BEDTIME;
              END
            ELSE LST_TIME := LST_TIME + CLOCK -
              NOTICE [ I ] .BEDTIME;
            I := NOTICE [ I ] .NEXT;
          UNTIL I = CURRENT;
          IF CANCEL_SIMULATION
          THEN
            BEGIN
              XSTR(' SIMULATION TERMINATES AT TIME',1,30);
              WRITE_REAL(CLOCK,10,0,REAL_TXT);
              XSTR( REAL_TXT,31,11);
              XSTR('(:0:)(:0:)',42,2);
              WRITE ( TRACE_DEVICE );
              "DISPLAY('SIMULATION FINISHED (:10:))';"
              CANCELLED := TRUE;
            END "FI";
          END "NEHT"
        ELSE DELAY := TRUE;
      END "CANCEL";
    
```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE UPDATE_TIME;
  VAR
    T , Y : INTEGER;
  BEGIN
    WITH NOTICE [ CURRENT ] DO
      BEGIN
        T := ATTRIBUTE_GET ;
        DELAY_LST [ T ] .ACTIVE_TIME :=
          DELAY_LST [ T ] .ACTIVE_TIME + OCCURRENCE - CLOCK;
        DELAY_LST [ T ] .LST_TIME := DELAY_LST [ T ] .LST_TIME +
          CLOCK - BEDTIME;

        IF MASTER_ID = T
          THEN OLD_TIMES := CLOCK;
        CLOCK := OCCURRENCE;
        NOTICE [ PREV ] .NEXT := NEXT;
        NOTICE [ NEXT ] .PREV := PREV;
        T := CURRENT;
        CURRENT := NEXT;
        NADD(T);
      END "HTIW";
    END "UPDATE_TIME";
  
```

```

PROCEDURE DMP_NOTICE ( HDR : INTEGER );
  VAR
    PTR : INTEGER;
  BEGIN
    PTR := HDR;
    XSTR(' OCCUR      DUMMY PTR ',1,20);
    WRITE ( TRACE_DEVICE );
    IF PTR <> 0
      THEN "DUMP OCCURRENCE, DUMMY, & PTR "
        REPEAT
          WITH NOTICE [ PTR ] DO
            BEGIN
              XSTR(' ',1,2);
              WRITE_REAL(OCCURRENCE,5,0,REAL_TXT);
              XSTR ( REAL_TXT,2,6);
              IF DUMMY
                THEN XSTR('TRUE ',11,6)
                ELSE XSTR('FALSE ',11,6);
              CHAR_STR(PTR,SM_TXT);
              XSTR(SM_TXT,17,4);
              WRITE(TRACE_DEVICE);
            END"WITH";
            PTR := NOTICE [ PTR ].NEXT;
          UNTIL (PTR = HDR ) OR ( PTR = 0 );
          FOR PTR := 1 TO 4 DO WRITE ( TRACE_DEVICE );
        END"DUMP_NOTICE";
    
```

APPENDIX : SEQUENTIAL VERSION CODE

```
PROCEDURE DMP_CNT;
  VAR
    I : INTEGER;
  BEGIN
    XSTR( ' PROCS DELAYED',1,14);
    DUMP_INT( PROCS_DELAYED,INT_TXT);
    XSTR( INT_TXT,15,6);
    XSTR( ' ACTIVE PROCS=',21,14);
    DUMP_INT(ACTIVE_PROCS,INT_TXT);
    XSTR(INT_TXT,35,6);
    WRITE( TRACE_DEVICE );
  END;"DMP_CNT"
```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE INSERT_TIME(T1: INTEGER);
  VAR
    X,Y,I: INTEGER;
    T : REAL;
BEGIN
  T := CONV(T1) + CLOCK;
  "%IF DEBUG"
  XSTR(' WAIT ',1,6);
  WRITE_REAL(T, 5,0,REAL_TXT);
  XSTR(REAL_TXT,7,6);
  WRITE( TRACE_DEVICE);
  DMP_NOTICE(CURRENT);
  "%END DEBUG"
  NGET(X);
  "%IF BUG"
  IF T = NOTICE [ CURRENT ].OCCURRENCE
    THEN DMP_NOTICE ( CURRENT);
  "%END BUG"
  IF T <= NOTICE [ CURRENT ] .OCCURRENCE
    THEN
      BEGIN
        Y := NOTICE [ CURRENT ] .PREV;
        CURRENT := X;
      END
    ELSE
      BEGIN
        I := TRUNC(T - LOWERBOUND) DIV DT;
        IF I >= N_INTVLS
          THEN I := N_INTVLS
          ELSE I := (ICURRENT + I) MOD N_INTVLS;
        Y := NOTICE [ INTVL_PTR [ I ] ] .PREV;
        WHILE NOTICE [ Y ] .OCCURRENCE > T DO Y := NOTICE [ Y ] .PREV;
      END "FI";
    WITH NOTICE [ X ] DO
      BEGIN
        OCCURRENCE := T;
        DUMMY := FALSE;
        ID := ATTRIBUTE_GET ;
        BEDTIME := CLOCK;
        NEXT := NOTICE [ Y ] .NEXT; PREV := Y;
        NOTICE [ NOTICE [ Y ] .NEXT ] .PREV := X;
        NOTICE [ Y ] .NEXT := X;
        I := ATTRIBUTE_GET ;
        SLEEP := SCENARIO_NUM;
        DELAY := TRUE;
      END "HTIW";
    "%IF DEBUG"
    DMP_NOTICE ( CURRENT );
    "%END DEBUG"
  END "INSERT_TIME";

```

APPENDIX : SEQUENTIAL VERSION CODE

```
PROCEDURE WAIT_TIME(T : INTEGER);
  BEGIN
    INSERT_TIME(T);
  END "WAIT_TIME";

PROCEDURE WAIT_RAND(T1 , T2 : INTEGER);
  VAR
    R : REAL;
    T : INTEGER;
  BEGIN
    IF T2 > T1
      THEN
        BEGIN
          RAND ( R );
          T := T1 + TRUNC(CONV(T2 - T1) * R);
          INSERT_TIME(T);
        END "FI";
      END "WAIT_RAND";
```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE EV_DELINK(P , POS : INTEGER);
"Adjust an event list by removing an entry;
zero out the entry removed"
BEGIN
  WITH DELAY_LST [ P ] .LINK [ POS ] DO
    BEGIN
      IF PREV <> 0
      THEN
        BEGIN
          DELAY_LST [ PREV ] .LINK [ PREV_POS ] .NEXT := NEXT;
          DELAY_LST[PREV].LINK[PREV_POS].NEXT_POS := NEXT_POS;
        END "FI";
      IF NEXT <> 0
      THEN
        BEGIN
          DELAY_LST [ NEXT ] .LINK [ NEXT_POS ] .PREV := PREV;
          DELAY_LST[NEXT].LINK[NEXT_POS].PREV_POS := PREV_POS;
        END "FI";
      WITH EVENT [ EVENT_NO ] DO
        BEGIN
          IF DELAYED & (Q_HEAD = P)
          THEN
            IF NEXT = 0
            THEN DELAYED := FALSE
            ELSE Q_HEAD := NEXT;
          Q := PRED(Q);
        END "WITH";
      PREV := 0;
      PREV_POS := 0;
      NEXT := 0;
      NEXT_POS := 0;
      EVENT_NO := 0;
    END "WITH";
  END "EV_DELINK";

```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE WAIT_SCALAR(I : INTEGER; VAR J , K : INTEGER);
VAR
  PAST,PAST_POS : INTEGER;
  DCNE : BOOLEAN;
BEGIN
  PAST := 0;
  PAST_POS := 0;
  WITH EVENT [ I ] DO
    BEGIN
      IF NOT DELAYED
      THEN BEGIN
        DELAYED := TRUE;
        Q_HEAD:= ATTRIBUTE_GET;
        J := 0;
        K := 0;
      END
      ELSE BEGIN
        J := Q_HEAD;
        DONE := FALSE;
        K := 1;
        WHILE DELAY_LST [ J ] .LINK [ K ] .EVENT_NO <> I DO
          K:=SUCC(K);
        REPEAT
          IF DELAY_LST [ J ] .LINK [ K ] .NEXT = 0
          THEN DONE := TRUE
          ELSE BEGIN
            PAST := J;
            PAST_POS := K;
            J:=DELAY_LST[PAST].LINK[PAST_POS].NEXT;
            K:=DELAY_LST[PAST].LINK[PAST_POS]
              .NEXT_POS;
          END "FI";
        UNTIL DONE;
      END "FI";
      NUM_WAITS := NUM_WAITS + 1.0;
      "Update Q"
      Q := SUCC(Q);
    END "HTIW";
  END "WAIT_SCALAR";

```


APPENDIX : SEQUENTIAL VERSION CODE

```
PROCEDURE AWAIT_SGNL(VAR P , EV : INTEGER);
```

```
VAR
```

```
  K : INTEGER;
```

```
BEGIN
```

```
  K := ATTRIBUTE_GET;
```

```
  SPEC1 [ K ] := P;
```

```
  SPEC2 [ K ] := EV;
```

```
  DELAY := TRUE;
```

```
  DELAY_LST [ K ].SLEEP := K;
```

```
END "AWAIT_SGNL";
```

```
PROCEDURE SPECIAL_EVENT; FORWARD;
```

```
PROCEDURE SPECIAL_EVENT;
```

```
  "This is needed to wake all scenarios waiting for an event"
```

```
VAR
```

```
  K , P , EV : INTEGER;
```

```
BEGIN
```

```
  K := ATTRIBUTE_GET;
```

```
  P := SPEC1 [ K ];
```

```
  EV := SPEC2 [ K ];
```

```
  WITH DELAY_LST [ K ] .LINK [ 1 ] DO
```

```
    BEGIN
```

```
      EV := EVENT_NO;
```

```
      P := NEXT;
```

```
      EV_DELINK( K , 1 );
```

```
      EVENT_NO := 0;
```

```
      PREV := 0;
```

```
      PREV_POS := 0;
```

```
      NEXT := 0;
```

```
      NEXT_POS := 0;
```

```
      DELAY_LST [ K ] .WAIT_TIME := DELAY_LST [ K ] .WAIT_TIME  
        + CLOCK - DELAY_LST [ K ] .REQ_TIME;
```

```
      DELAY_LST [ K ] .REQ_TIME := 0.0;
```

```
    END "HTIW";
```

```
    IF P <> 0
```

```
      THEN BEGIN
```

```
        ACTIVE_PROCESSES [ ARRIVAL ] := SCENARIO_NUM;
```

```
        SCENARIO_NUM := DELAY_LST [ P ].SLEEP;
```

```
        SPECIAL_EVENT;
```

```
      END;
```

```
END "SPECIAL_EVENT";
```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE WAIT_EVENT(I : INTEGER);
VAR
  PAST , PAST_POS , K : INTEGER;
BEGIN
  WAIT_SCALAR(I,PAST,PAST_POS);
  K := ATTRIBUTE_GET;
  IF TRACE_EVENT
    THEN BEGIN
      WRITE_PREFACE;
      XSTR('AWAITS EVENT ',24,14);
      DUMP_INT(I,INT_TXT);
      XSTR (INT_TXT,38,6);
      XSTR('(:0:)(:0:)',44,2);
      WRITE ( TRACE_DEVICE );
      END"FI";
    IF PAST <> 0
      THEN WITH DELAY_LST [ PAST ] .LINK [ PAST_POS ] DO
        BEGIN
          NEXT := K;
          NEXT_POS := 1;
          END "FI";
        WITH DELAY_LST [ K ] .LINK [ 1 ] DO
          BEGIN
            DELAY_LST [ K ] .REQ_TIME := CLOCK;
            EVENT_NO := I;
            PREV := PAST;
            PREV_POS := PAST_POS;
            END "WITH";
          AWAIT_SGNL(K,PAST);
        END "WAIT_EVENT";

```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE WAIT_VECTOR(VAR I : INTEGER ; VEC : EVENT_VEC);
VAR
  PAST , PAST_POS , J , K , L : INTEGER;
BEGIN
  K := ATTRIBUTE_GET;
  I := 1;
  FOR J := 1 TO MAX_EVENT DO
    IF VEC [ J ]
      THEN BEGIN
        WAIT_SCALAR(J,PAST,PAST_POS);
        WITH DELAY_LST [ K ] .LINK [ I ] DO
          "If range error here then EVENT_VEC has >
          PMAX_EVENT entries set"
          BEGIN
            EVENT_NO := J;
            PREV := PAST;
            PREV_POS := PAST_POS;
          END;
          IF PAST <> 0
            THEN WITH DELAY_LST [ PAST ] .LINK [ PAST_POS ] DO
              BEGIN
                NEXT := K;
                NEXT_POS := I;
              END;
              I := SUCC(I);
            END "FI";
          DELAY_LST [ K ] .REQ_TIME := CLOCK;
          IF TRACE_EVENT
            THEN BEGIN
              WRITE_PREFACE;
              XSTR('AWAITS EVENTS ',24,14);
              L := 38;
              FOR J := 1 TO MAX_EVENT DO
                IF VEC [ J ]
                  THEN BEGIN
                    DUMP_INT(J,INT_TXT);
                    XSTR(INT_TXT,L,6);
                    XSTR(' ',L+6,2);
                    L := L + 8;
                  END"FI";
                    XSTR('(:0:)(:0:)',L,2);
                    WRITE(TRACE_DEVICE);
                  END "FI";
                AWAIT_SGNL(K,I);
              END "WAIT_VECTOR";

```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE EV_RESET(EV : INTEGER);
VAR
  P , K : INTEGER;
  "An event has occurred; follow that event list &
  remove processes on it from any other event lists
  they may be on"
BEGIN
  P := EVENT [ EV ] .Q_HEAD;
  REPEAT
    WITH DELAY_LST [ P ] DO
      BEGIN
        IF LINK [ 1 ] .EVENT_NO <> EV
          THEN EV_DELINK(P,1);
        K := 2;
        WHILE (K <= PMAX_EVENT) DO
          BEGIN
            IF LINK [ K ] .EVENT_NO <> 0
              THEN IF LINK [ K ] .EVENT_NO = EV
                THEN WITH LINK [ K ] DO
                  BEGIN
                    LINK [ 1 ] := LINK [ K ] ;
                    PREV := 0;
                    PREV_POS := 0;
                    NEXT := 0;
                    NEXT_POS := 0;
                    EVENT_NO := 0;
                  END "WITH"
                ELSE EV_DELINK(P,K);
            K := SUCC(K);
          END"WHILE";
        END "HTIW";
        P := DELAY_LST [ P ] .LINK [ 1 ] .NEXT;
      UNTIL P = 0;
    END "EV_RESET";
  
```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE SIGNAL(EV : INTEGER);
BEGIN
  IF TRACE_EVENT
    THEN BEGIN
      WRITE_PREFACE;
      XSTR('SIGNALS ',24,8);
      DUMP_INT(EV,INT_TXT);
      XSTR(INT_TXT,32,6);
      XSTR('(:0:)(:0:)',38,2);
      WRITE( TRACE_DEVICE );
    END "FI";
  WITH EVENT [ EV ] DO
    BEGIN
      "update statistics"
      NUM_SIGNALS := NUM_SIGNALS + 1.0;
      "calculate average Q length at time of a SIGNAL"
      Q_AVE := (Q_AVE * (NUM_SIGNALS - 1.0) + CONV(Q))/NUM_SIGNALS;
      IF Q > Q_MAX
        THEN Q_MAX := Q;
      "average maximum delay time"
      IF DELAYED
        THEN BEGIN
          AVE_WAIT := (AVE_WAIT * (NUM_SIGNALS - 1.0) + CLOCK
            - DELAY_LST [ Q_HEAD ] .REQ_TIME)/ NUM_SIGNALS;
          EV_RESET(EV);
          ACTIVE_PROCESSES [ ARRIVAL ] := SCENARIO_NUM;
          SCENARIO_NUM := DELAY_LST [ Q_HEAD ] .SLEEP;
          SPECIAL_EVENT;
        END "FI";
      END "HTIW";
    END "SIGNAL";
  END

```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE DCL_FACILITIES(N : INTEGER; MAXI : MAX_FAC);
VAR
  FI , FJ , I : INTEGER;
  STR : INT_STR;
BEGIN
  FACILITY_INDEXER [ 0 ] := 1;
  "%IF DEBUG"
  XSTR(' FACILITIES ',1,12);
  WRITE(TRACE_DEVICE);
  DMP_CNT;
  "%END DEBUG"
  FOR FI := 1 TO N-1 DO
    BEGIN
      FACILITY_INDEXER [ FI ] := 1;
      FOR FJ := 0 TO (FI - 1) DO
        FACILITY_INDEXER [ FI ] := FACILITY_INDEXER [ FI ]
          + MAXI [ FJ ];
      END"FOR";
    FOR FI := N TO FACILITY_TYPES DO FACILITY_INDEXER [ FI ] :=0;
    "%IF DEBUG"
    FOR I := 0 TO N DO
      BEGIN
        DUMP_INT(FACILITY_INDEXER[I],STR);
        XSTR(STR,(10*I + 1),6);
      END;
    WRITE ( TRACE_DEVICE );
    "%END DEBUG"
    "initialize facility array"
    FCOUNT := FACILITY_INDEXER [ N-1 ] + MAXI [ N-1 ] - 1;
    FOR I := 1 TO FCOUNT DO
      WITH FACILITY [ I ] DO
        BEGIN
          FREE := TRUE;
          Q_HEAD := 0;
          RLSE_CNT := 0.0;
          USE_TIME := 0.0;
          Q := 0;
          Q_AVE := 0.0;
          Q_MIN := MAX_PROCESS;
          Q_MAX := 0;
          SEIZE_TIME := 0.0;
          USE_CNT := 0.0;
          AVE_WAIT := 0.0;
        END "WITH FACILITY[I]";
      END "DCL_FCLTS";
    
```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE SEIZE(NAME : UNIV INTEGER; I : UNIV INTEGER);
VAR
  J , K , FINDEX: INTEGER;
  NOT_DELAYED : BOOLEAN;
BEGIN
  FINDEX := FACILITY_INDEXER [ NAME ] + I -1;
  FACILITY[FINDEX].USE_CNT:=FACILITY[FINDEX].USE_CNT+1.0;
  K := ATTRIBUTE_GET;
  DELAY_LST [ K ] .REQ_TIME := CLOCK;
  NOT_DELAYED := TRUE;
  WITH FACILITY [ FINDEX ] DO
    IF NOT FACILITY [ FINDEX ] .FREE
      THEN BEGIN
        FACILITY[FINDEX].Q := SUCC(FACILITY [ FINDEX ] .Q);
        "Update Q_AVE, Q_MIN,Q_MAX"
        Q_AVE := ((USE_CNT - 1.0) * Q_AVE+CONV(Q)) / USE_CNT;
        IF Q < Q_MIN
          THEN Q_MIN := Q
          ELSE IF Q > Q_MAX
            THEN Q_MAX := Q;
        IF TRACE_FACILITY
          THEN BEGIN
            WRITE_PREFACE;
            XSTR('IS DELAYED UPON (:35:)',24,18);
            DUMP_INT(I,INT_TXT);
            XSTR (INT_TXT,42,6);
            XSTR('OF FACILITY TYPE ',49,18);
            DUMP_INT(NAME,INT_TXT);
            XSTR(INT_TXT,67,6);
            XSTR('(:0:)(:0:)',73,2);
            WRITE(TRACE_DEVICE);
          END "FI";
        IF FACILITY [ FINDEX ] .Q_HEAD = 0
          THEN FACILITY [ FINDEX ] .Q_HEAD := K
          ELSE BEGIN
            J := FACILITY [ FINDEX ] .Q_HEAD;
            WHILE DELAY_LST [J].LINK [1].NEXT <> 0 DO
              J := DELAY_LST [J] .LINK [1] .NEXT;
            DELAY_LST [ J ] .LINK [ 1 ] .NEXT := K;
          END;
        DELAY_LST [ K ] .LINK [ 1 ] .EVENT_NO := FINDEX;
        SPEC1 [ K ] := J;
        SPEC2 [ K ] := I;
        SPEC3 [ K ] := NAME;
        SPEC4 [ K ] := NOT_DELAYED;
        DELAY := TRUE;
        DELAY_LST [ K ] .SLEEP := K;
      END "FI" "WITH";
    END IF;
  END WITH;
END SEIZE;

```

APPENDIX : SEQUENTIAL VERSION CODE

```

IF NOT DELAY
THEN
BEGIN
  WITH FACILITY [ FINDEX ] DO
  BEGIN
    SEIZE_TIME := CLOCK;
    "Update Q_AVE,Q_MIN,Q_MAX"
    IF NOT_DELAYED
      THEN Q_AVE := ((USE_CNT-1.0) * Q_AVE+CONV(Q))/ USE_CNT;
    IF Q < Q_MIN
      THEN Q_MIN := Q
      ELSE IF Q > Q_MAX
        THEN Q_MAX := Q;
    AVE_WAIT := (AVE_WAIT * (USE_CNT - CONV(Q) - 1.0) +
      CLOCK-DELAY_LST[K].REQ_TIME)/(USE_CNT-CONV(Q));
    FREE := FALSE;
  END "WITH";
  IF TRACE_FACILITY
  THEN BEGIN
    WRITE_PREFACE;
    XSTR('SEIZES (:35:)',24,8);
    DUMP_INT(I,INT_TXT);
    XSTR(INT_TXT,32,6);
    XSTR('OF FACILITY TYPE ',39,18);
    DUMP_INT(NAME,INT_TXT);
    XSTR(INT_TXT,57,6);
    XSTR('(:0:)(:0:)',63,2);
    WRITE(TRACE_DEVICE);
  END "FI";
  DELAY_LST [ K ] .REQ_TIME := 0.0;
  END;"DELAY IF"
END "SEIZE";

```


APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE SPECIAL_SEIZE;
  "Split from SEIZE to allow for delaying processes"
VAR
  J , K , I , NAME , FINDEX: INTEGER;
  NOT_DELAYED : BOOLEAN;
BEGIN
  K := SCENARIO_NUM;
  J := SPEC1 [ K ];
  I := SPEC2 [ K ];
  NAME := SPEC3 [ K ];
  NOT_DELAYED := SPEC4 [ K ];
  K := ATTRIBUTE_GET;
  DELAY_LST [ K ] .WAIT_TIME := DELAY_LST [ K ] .WAIT_TIME
    + CLOCK - DELAY_LST [ K ] .REQ_TIME;
  FINDEX := DELAY_LST [ K ] .LINK [ 1 ] .EVENT_NO;
  NOT_DELAYED := FALSE;
  DELAY_LST [ K ] .LINK [ 1 ] .NEXT := 0;
  DELAY_LST [ K ] .LINK [ 1 ] .EVENT_NO := 0;
  WITH FACILITY [ FINDEX ] DO
    BEGIN
      SEIZE_TIME := CLOCK;
      "Update Q_AVE, Q_MIN, Q_MAX"
      IF NOT_DELAYED
        THEN Q_AVE := ((USE_CNT-1.0) * Q_AVE+CONV(Q))/ USE_CNT;
      IF Q < Q_MIN
        THEN Q_MIN := Q
      ELSE IF Q > Q_MAX
        THEN Q_MAX := Q;
      AVE_WAIT := (AVE_WAIT * (USE_CNT - CONV(Q) - 1.0) +
        CLOCK-DELAY_LST[K].REQ_TIME)/(USE_CNT-CONV(Q));
      FREE := FALSE;
    END "WITH";
  IF TRACE_FACILITY
    THEN BEGIN
      WRITE_PREFACE;
      XSTR('SEIZES (:35:)',24,8);
      DUMP_INT(I,INT_TXT);
      XSTR(INT_TXT,32,6);
      XSTR('OF FACILITY TYPE ',39,18);
      DUMP_INT(NAME,INT_TXT);
      XSTR(INT_TXT,57,6);
      XSTR('(:0:)(:0:)',63,2);
      WRITE(TRACE_DEVICE);
    END "FI";
  DELAY_LST [ K ] .REQ_TIME := 0.0;
END "SPECIAL_SEIZE";

```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE RELEASE(NAME : UNIV INTEGER; I : UNIV INTEGER);
  VAR
    J , FINDEX : INTEGER;
  BEGIN
    FINDEX := FACILITY_INDEXER [ NAME ] + I - 1;
    WITH FACILITY [ FINDEX ] DO
      BEGIN
        USE_TIME := USE_TIME + CLOCK - SEIZE_TIME;
        SEIZE_TIME := 0.0;
        RLSE_CNT := RLSE_CNT + 1.0;
        IF TRACE_FACILITY
          THEN BEGIN
            WRITE_PREFACE;
            XSTR('RELEASES (:35:)',24,10);
            DUMP_INT(I,INT_TXT);
            XSTR(INT_TXT,34,6);
            XSTR('OF FACILITY TYPE ',41,18);
            DUMP_INT(NAME,INT_TXT);
            XSTR(INT_TXT,59,6);
            XSTR('(:0:)(:0:)',65,2);
            WRITE(TRACE_DEVICE);
          END "FI";
        IF Q_HEAD <> 0
          THEN BEGIN
            J := Q_HEAD;
            Q_HEAD := DELAY_LST [ Q_HEAD ] .LINK [ 1 ] .NEXT;
            Q := PRED(Q);
            ACTIVE_PROCESSES [ ARRIVAL ] := SCENARIO_NUM;
            SCENARIO_NUM := DELAY_LST [ J ] .SLEEP;
            SPECIAL_SEIZE;
          END "FI"
        ELSE FREE := TRUE;
      END "HTIW";
    END "RELEASE";
  END "RELEASE";

PROCEDURE EVAL(COND : INTEGER; VAR VALUE : BOOLEAN);
  BEGIN
    CASE COND OF
      1,2,3 : VALUE := CNTR [ COND ] .VALUE > 0
    END "ESAC";
  END "EVAL";

```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE CHK_CDTNS;
  VAR
    EOL , FOUND , VAL , ON_HEAD : BOOLEAN;
  BEGIN
    FOUND := FALSE;
    EOL := FALSE;
    ON_HEAD := TRUE;
    REPEAT
      IF CDTN_PTR = 0
        THEN EOL := TRUE
        ELSE EVAL(TRUNC(NOTICE [ CDTN_PTR ] .OCCURRENCE),VAL);
      IF NOT EOL
        THEN IF VAL
              THEN FOUND := TRUE
              ELSE IF (CDTN_PTR = CDTN_LST) & NOT ON_HEAD
                    THEN EOL := TRUE
                    ELSE CDTN_PTR := NOTICE [ CDTN_PTR ] .NEXT;
        ON_HEAD := FALSE;
      UNTIL (EOL OR FOUND);
      IF NOT FOUND
        THEN CDTN_PTR := 0;
    END "CHK_CDTNS";

```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE WAIT_UNTIL(C : INTEGER);
  VAR
    VAL: BOOLEAN;
    I , K : INTEGER;
  BEGIN
    EVAL(C,VAL);
    IF NOT VAL
      THEN BEGIN
        IF TRACE_CNTR
          THEN BEGIN
            WRITE_PREFACE;
            XSTR('IS DELAYED ON CONDITION ',24,24);
            CHAR_STR(C,SM_TXT);
            XSTR(SM_TXT,48,4);
            XSTR('(:0:)(:0:)',52,2);
            WRITE(TRACE_DEVICE);
          END"FI";
        NGET(K);
        WITH NOTICE [ K ] DO
          BEGIN
            IF CDTN_LST <> 0
              THEN BEGIN
                NEXT := CDTN_LST;
                PREV := NOTICE [ CDTN_LST ] .PREV;
                NOTICE [ CDTN_LST ] .PREV := K;
                NOTICE [ PREV ] .NEXT := K;
              END "THEN"
            ELSE BEGIN
                CDTN_LST := K;
                PREV := K;
                NEXT := K;
              END "ELSE";
            OCCURRENCE := CONV(C);
            DELAY_LST [ ATTRIBUTE_GET ] .REQ_TIME := CLOCK;
          END "WITH";
          I:= SCENARIO_NUM;
          SPEC1 [ K ] := K;
          SPEC2 [ K ] := C;
          SPEC4 [ K ] := VAL;
          DELAY := TRUE;
          DELAY_LST [ K ] . SLEEP := K;
        END "FI";
      END "WAIT_UNTIL";

```

APPENDIX : SEQUENTIAL VERSION CODE

PROCEDURE SPECIAL_UNTIL; FORWARD;

PROCEDURE SPECIAL_UNTIL;

"Split from WAIT_UNTIL to allow for continuing processes"

VAR

VAL: BOOLEAN;

I , K , C : INTEGER;

BEGIN

I := SCENARIO_NUM;

K := SPEC1 [I];

C := SPEC2 [I];

VAL := SPEC4 [I];

"This is where the condition in NOTICE [CDTN_PTR]
becomes true & is CONTINUED"

K := CDTN_PTR;

I := ATTRIBUTE_GET;

DELAY_LST [I].WAIT_TIME := DELAY_LST [I].WAIT_TIME +
CLOCK - DELAY_LST [I] .REQ_TIME;

DELAY_LST [I] .REQ_TIME := 0.0;

WITH NOTICE [CDTN_PTR] DO

IF CDTN_PTR = NEXT

THEN BEGIN

CDTN_LST := 0;

CDTN_PTR := 0;

END

ELSE BEGIN

NOTICE [NEXT] .PREV := PREV;

NOTICE [PREV] .NEXT := NEXT;

IF CDTN_LST = CDTN_PTR

THEN CDTN_LST := NEXT;

CDTN_PTR := NEXT;

END "FI";

NADD(K);

IF CDTN_PTR <> 0

THEN BEGIN

ACTIVE_PROCESSES [ARRIVAL] := SCENARIO_NUM;

SCENARIO_NUM := NOTICE [CDTN_PTR] .SLEEP;

SPECIAL_UNTIL;

END;

END "SPECIAL_UNTIL";

APPENDIX : SEQUENTIAL VERSION CODE

```
PROCEDURE DCL_COUNTERS(CMAX : INTEGER);
```

```
  VAR
```

```
    I: INTEGER;
```

```
  BEGIN
```

```
    FOR I := 1 TO CMAX DO WITH CNTR [ I ] DO
```

```
      BEGIN
```

```
        VALUE := 0;
```

```
        AVE_VALUE := 0.0;
```

```
        MIN_VALUE := 32767;
```

```
        MAX_VALUE := 0;
```

```
        USE_CNT := 0.0;
```

```
      END "HTIW";
```

```
    CNTR_LIMIT := CMAX;
```

```
  END "DCL_CNTRS";
```

```
PROCEDURE CNTR_STATS(I : INTEGER);
```

```
  BEGIN
```

```
    WITH CNTR [ I ] DO
```

```
      BEGIN
```

```
        AVE_VALUE := (USE_CNT * AVE_VALUE + CONV(VALUE)) / (USE_CNT + 1.0);
```

```
        USE_CNT := USE_CNT + 1.0;
```

```
        IF VALUE < MIN_VALUE
```

```
          THEN MIN_VALUE := VALUE
```

```
          ELSE IF VALUE > MAX_VALUE
```

```
            THEN MAX_VALUE := VALUE;
```

```
      END "HTIW";
```

```
  END "CNTR_STATS";
```

```
FUNCTION TEST(C : INTEGER) : INTEGER;
```

```
  BEGIN
```

```
    TEST := CNTR [ C ] .VALUE;
```

```
  END "TEST";
```

```
PROCEDURE ASSIGN_COUNT(I , VAL : INTEGER);
```

```
  BEGIN
```

```
    "Update statistics"
```

```
    CNTR_STATS(I);
```

```
    "or write out statistics for CNTR [ I ] ???"
```

```
    CNTR [ I ] .VALUE := VAL;
```

```
  END "ASSIGN_CNT";
```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE INC_COUNT(I , AMT : INTEGER);
BEGIN
    WITH CNTR [ I ] DO
        BEGIN
            VALUE := VALUE + AMT;
            "Update statistics"
            CNTR_STATS(I);
            IF TRACE_CNTR
                THEN BEGIN
                    WRITE_PREFACE;
                    XSTR('INCREMENTS COUNTER',24,18);
                    DUMP_INT(I,INT_TXT);
                    XSTR(INT_TXT,42,6);
                    XSTR (' BY ',48,4);
                    DUMP_INT(AMT,INT_TXT);
                    XSTR(INT_TXT,52,6);
                    XSTR('(:0:)(:0:)',58,2);
                    WRITE(TRACE_DEVICE);
                END "FI";
            CDTN_PTR := CDTN_LST;
            CHK_CDTNS;
            IF CDTN_PTR <> 0
                THEN BEGIN
                    ACTIVE_PROCESSES[ARRIVAL] := SCENARIO_NUM;
                    SCENARIO_NUM := NOTICE [ CDTN_PTR ].SLEEP;
                    SPECIAL_UNTIL;
                END;
            END "HTIW";
        END "INC_CNT";
END "INC_CNT";

FUNCTION TIME : REAL;
BEGIN
    TIME := CLOCK;
END "TIME";

PROCEDURE TRACE(F1, C, E: BOOLEAN);
BEGIN
    TRACE_FACILITY := F1;
    TRACE_CNTR      := C;
    TRACE_EVENT     := E;
END "TRACE";

PROCEDURE TRACE_FILE(NAME: FNAME);
BEGIN
    TRACE_TABLE [ ATTRIBUTE_GET ] := NAME;
END "TRACE_FILE";

```

APPENDIX : SEQUENTIAL VERSION CODE

```
PROCEDURE NEW_PAGE(DEV: FNAME);
```

```
  BEGIN
```

```
    XSTR('(:12:) ',1,2);
```

```
    WRITE ( DEV );
```

```
  END "NEW_PAGE";
```

```
PROCEDURE WRITE_TIME(DEV: FNAME);
```

```
  BEGIN
```

```
    XSTR(' THE CURRENT TIME IS ',1,22);
```

```
    WRITE_REAL(CLOCK,10,0,REAL_TXT);
```

```
    XSTR(REAL_TXT,23,11);
```

```
    XSTR ( '(:0:)(:0:)',34,2);
```

```
    WRITE(DEV);
```

```
  END "WRITE_TIME";
```

```
PROCEDURE DISPLAY_TIME;
```

```
  BEGIN
```

```
    WRITE_TIME(PRINTR);
```

```
  END "DISPLAY_TIME";
```


APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE REPORT_FACS(DEV : FNAME);
  VAR
    KIND , INST , I : INTEGER;
  BEGIN "Write out heading"
    NEW_PAGE(DEV);
    WRITE_TIME(DEV);
    XSTR('      TYPE  INSTANCE  TIME HELD ',1,30);
    XSTR('  AVE Q LENGTH    MAX Q   MIN Q   ',31,32);
    XSTR('AVE DELAY TIME      (:35:)SEIZES  ',63,28);
    XSTR('(:35:)RLSES(:0:)(:0:)',97,8);
    WRITE(DEV);
    KIND := 0;
    INST := 1;
    FOR I := 1 TO FCOUNT DO WITH FACILITY [ I ] DO
      BEGIN
        IF KIND <> (FACILITY_TYPES - 1)
          THEN IF I = FACILITY_INDEXER [ KIND + 1 ]
            THEN BEGIN
              KIND := SUCC( KIND );
              INST := 1;
              END "FI";
            XSTR(' ',1,2);
            DUMP_INT(KIND,INT_TXT);
            XSTR ( INT_TXT,3,6);
            DUMP_INT(INST,INT_TXT);
            XSTR(INT_TXT,12,6);
            WRITE_REAL(USE_TIME,8,0,REAL_TXT);
            XSTR(REAL_TXT,21,9);
            WRITE_REAL(Q_AVE,3,2,REAL_TXT);
            XSTR(REAL_TXT,36,7);
            DUMP_INT(Q_MAX,INT_TXT);
            XSTR(INT_TXT,47,6);
            DUMP_INT(Q_MIN,INT_TXT);
            XSTR(INT_TXT,55,6);
            WRITE_REAL(AVE_WAIT,6,3,REAL_TXT);
            XSTR(REAL_TXT,66,11);
            WRITE_REAL(USE_CNT,8,0,REAL_TXT);
            XSTR(REAL_TXT,80,9);
            WRITE_REAL(RLSE_CNT,8,0,REAL_TXT);
            XSTR(REAL_TXT,94,9);
            XSTR('(:0:)(:0:)',103,2);
            WRITE(DEV);
            INST := SUCC(INST);
          END"HTIW";
        END "REPORT_FACS";
      END
    END
  END

```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE REPORT_COUNTERS(DEV: FNAME);
  VAR
    I: INTEGER;
  BEGIN "Write out heading"
    NEW_PAGE(DEV);
    WRITE_TIME(DEV);
    XSTR(' COUNTER ',1,10);
    XSTR('TIMES SET ',15,10);
    XSTR('AVE VALUE ',30,10);
    XSTR('MAX VALUE ',45,10);
    XSTR('MIN VALUE (:0:)(:0:)',60,12);
    WRITE(DEV);
    FOR I := 1 TO CNTR_LIMIT DO WITH CNTR [ I ] DO
      BEGIN
        XSTR(' ',1,2);
        DUMP_INT(I,INT_TXT);
        XSTR(INT_TXT,4,6);
        WRITE_REAL(USE_CNT,8,0,REAL_TXT);
        XSTR(REAL_TXT,16,9);
        WRITE_REAL(AVE_VALUE,7,3,REAL_TXT);
        XSTR(REAL_TXT,28,12);
        DUMP_INT(MAX_VALUE,INT_TXT);
        XSTR(INT_TXT,47,6);
        DUMP_INT(MIN_VALUE,INT_TXT);
        XSTR(INT_TXT,62,6);
        XSTR('(:0:)(:0:)',68,2);
        WRITE(DEV);
      END "HTIW";
    END "REPORT_COUNTERS";
  
```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE REPORT_EVENTS(DEV : FNAME);
VAR
  I : INTEGER;
BEGIN "Write heading"
  NEW_PAGE(DEV);
  WRITE_TIME(DEV);
  XSTR(' EVENT ',1,8);
  XSTR('(:35:) SIGNALS ',15,10);
  XSTR('(:35:) WAITS ',30,8);
  XSTR('AVE Q LENGTH',45,12);
  XSTR('MAX Q ',65,6);
  XSTR('AVE DELAY TIME(:0:)(:0:)',75,16);
  WRITE(DEV);
  FOR I := 1 TO MAX_EVENT DO WITH EVENT [ I ] DO
    BEGIN
      XSTR(' ',1,2);
      DUMP_INT(I,INT_TXT);
      XSTR(INT_TXT,3,6);
      WRITE_REAL(NUM_SIGNALS,8,0,REAL_TXT);
      XSTR(REAL_TXT,15,9);
      WRITE_REAL(NUM_WAITS,8,0,REAL_TXT);
      XSTR(REAL_TXT,29,9);
      WRITE_REAL(Q_AVE,7,3,REAL_TXT);
      XSTR(REAL_TXT,45,12);
      DUMP_INT(Q_MAX,INT_TXT);
      XSTR(INT_TXT,64,6);
      WRITE_REAL(AVE_WAIT,6,3,REAL_TXT);
      XSTR(REAL_TXT,80,11);
      XSTR('(:0:)(:0:)',91,2);
      WRITE(DEV);
    END "HTIW";
  END "REPORT EVENTS";

```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE REPORT_PROCESSES(DEV : FNAME);
VAR
  I : INTEGER;
BEGIN "Write heading"
  NEW_PAGE(DEV);
  WRITE_TIME(DEV);
  XSTR('ADVANCES(:0:)(:0:)',33,10);
  XSTR(' UNITS SUSPENDED (:0:)(:0:)',41,18);
  WRITE(DEV);
  XSTR(' PROCESS',1,8);
  XSTR('DELAY TIME',15,10);
  XSTR('CLOCK BY(:0:)(:0:)',33,10);
  XSTR(' ON TIME LIST (:0:)(:0:)',41,16);
  WRITE(DEV);
  FOR I := SCENARIO1 TO MAX_PROCESS DO
    WITH DELAY_LST [ I ] DO
      BEGIN
        DUMP_INT(I,INT_TXT);
        XSTR(INT_TXT,1,6);
        WRITE_REAL(WAIT_TIME,10,0,REAL_TXT);
        XSTR(REAL_TXT,14,11);
        WRITE_REAL(ACTIVE_TIME,10,0,REAL_TXT);
        XSTR(REAL_TXT,30,11);
        XSTR('(:0:)(:0:)',41,2);
        WRITE_REAL(LST_TIME,10,0,REAL_TXT);
        XSTR(REAL_TXT,41,11);
        XSTR('(:0:)(:0:)',52,2);
        WRITE(DEV);
      END"HTIW";
    END "REPORT_PROCESSES";
  END

```

APPENDIX : SEQUENTIAL VERSION CODE

" INSERT SCENARIO PROCEDURES HERE "

```
PROCEDURE INCR ( N : INTEGER );
  BEGIN
    INSTR_PTR [ N ] := SUCC ( INSTR_PTR [ N ] );
  END;
```

```
PROCEDURE NULLPROC( N : INTEGER ; VAR PARAM : SIM_INSTR_REC );
  BEGIN
    WITH PARAM DO
      CASE INSTR_PTR [ N ] OF
        1: BEGIN
            SIM_MON_INSTR := 1;
            INCR ( N );
          END;
        2: BEGIN
            SIM_MON_INSTR := 2;
            A := FALSE;
            INCR ( N );
          END;
      END "CASE";
    END "NULLPROC";
```

```
PROCEDURE S_JOB1_CENTER ( LINE : PRT_LINE;
                          START ,
                          COUNT ,
                          N : INTEGER );

  VAR
    I , J : INTEGER;
  BEGIN
    FOR I := 1 TO COUNT DO
      BEGIN
        J := START + ( I - 1 );
        S_JOB1_TEXT [ N , J ] := LINE [ I ] ;
      END;
    END; " S_JOB1_CENTER"
```

```
PROCEDURE S_JOB2_CENTER ( LINE : PRT_LINE;
                          START ,
                          COUNT ,
                          N : INTEGER );

  VAR
    I , J : INTEGER;
  BEGIN
    FOR I := 1 TO COUNT DO
      BEGIN
        J := START + ( I - 1 );
        S_JOB2_TEXT [ N , J ] := LINE [ I ] ;
      END;
    END; "S_JOB2_CENTER"
```

APPENDIX : SEQUENTIAL VERSION CODE

"This converts enumerations to integers "

```
FUNCTION ENUM_TO_INTEGER ( N : UNIV SHORTINTEGER ) : INTEGER;  
  BEGIN  
    ENUM_TO_INTEGER := N;  
  END;
```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE JOB1TEST( N : INTEGER; VAR PARAM : SIM_INSTR_REC;
                   INSTANCE : INTEGER );
BEGIN
  WITH PARAM DO
    CASE INSTR_PTR [ N ] OF
      1: BEGIN
          SIM_MON_INSTR := 1;
          "ALIVE"
          INCR ( N );
        END;
      2: BEGIN
          SIM_MON_INSTR := 16;
          E := 2;
          F := 1;
          "INC_COUNT"
          INCR ( N );
        END;
      3: BEGIN
          SIM_MON_INSTR := 9;
          E := ENUM_TO_INTEGER ( S_CPU );
          F := 1;
          "SEIZE"
          INCR ( N );
        END;
      4: BEGIN
          S_JOB1_CENTER ( ' JOB1 STARTS SLICE AND JOB TIME IS'
                        , 10, 34, INSTANCE );
          SIM_MON_INSTR := 26;
          E := S_JOB1_JOB_TIME [ INSTANCE ];
          "INT_CHAR"
          INCR ( N );
        END;
      5: BEGIN
          FOR III := 1 TO 6 DO
            S_JOB1_TEXT [ INSTANCE , 43 + III ] :=
              L [ III ];
            SIM_MON_INSTR := 25;
            FOR III := 1 TO 130 DO
              K [ III ] := S_JOB1_TEXT [ INSTANCE , III ];
              "WRITE"
              INCR ( N );
            END;
          END;
      6: BEGIN
          FOR III := 1 TO 130 DO
            S_JOB1_TEXT [ INSTANCE , III ] := BLANK;
            IF S_JOB1_JOB_TIME [ INSTANCE ] <= S_JOB1_SLICE
              THEN S_JOB1_DELTA [ INSTANCE ] :=
                S_JOB1_JOB_TIME [ INSTANCE ]
              ELSE S_JOB1_DELTA [ INSTANCE ] := S_JOB1_SLICE;
            SIM_MON_INSTR := 5;
            E := S_JOB1_DELTA [ INSTANCE ];
            "WAIT_TIME"
            INCR ( N );
          END;
    END;
  END;
END;

```

APPENDIX : SEQUENTIAL VERSION CODE

```

7: BEGIN
    "Calculate remaining job time"
    S_JOB1_JOB_TIME [ INSTANCE ] := S_JOB1_JOB_TIME
        [ INSTANCE ] - S_JOB1_DELTA [ INSTANCE ];
    SIM_MON_INSTR := 30;
    "NO OPERATION"
    INCR ( N );
END;
8: BEGIN
    SIM_MON_INSTR := 16;
    E := 2;
    F := -1;
    "INC_COUNT"
    INCR ( N );
END;
9: BEGIN
    S_JOB1_CENTER ( ' JOB1 ENDS SLICE AND JOBTIME IS '
        ,10,32,INSTANCE);
    SIM_MON_INSTR := 26;
    E := S_JOB1_JOB_TIME [ INSTANCE ];
    "INT_CHAR"
    INCR ( N );
END;
10: BEGIN
    FOR III := 1 TO 6 DO
        S_JOB1_TEXT [ INSTANCE , 43 + III ] :=
            L [ III ];
    SIM_MON_INSTR := 25;
    FOR III := 1 TO 130 DO
        K [ III ] := S_JOB1_TEXT [ INSTANCE , III ];
    "WRITE"
    INCR ( N );
END;
11: BEGIN
    FOR III := 1 TO 130 DO
        S_JOB1_TEXT [ INSTANCE , III ] := BLANK;
        "Blank out print line"
    IF S_JOB1_JOB_TIME [ INSTANCE ] <= 0
        "i. e. if job step done "
        THEN INCR ( N ) "proceed to I-O"
        ELSE INSTR_PTR [ N ] := 2; "return to SEIZE"
    SIM_MON_INSTR := 10;
    E := ENUM_TO_INTEGER ( S_CPU );
    F := 1;
    " RELEASE"
END;
12: BEGIN
    SIM_MON_INSTR := 16;
    E := 1;
    F := 1;
    "INC_COUNT"
    INCR ( N );
END;

```


APPENDIX : SEQUENTIAL VERSION CODE

```

13: BEGIN
    SIM_MON_INSTR := 9;
    E := ENUM_TO_INTEGER ( S_DISK);
    F := S_JOB1_INDEX ;
    "SEIZE"
    INCR ( N );
END;
14: BEGIN
    S_JOB1_CENTER ('JOB1 STARTS I/O'
        ,1,15,INSTANCE);
    SIM_MON_INSTR := 25;
    FOR III := 1 TO 130 DO
        K [ III ] := S_JOB1_TEXT [ INSTANCE , III ];
        "WRITE"
        INCR ( N ) ;
    END;
15: BEGIN
    FOR III := 1 TO 130 DO
        S_JOB1_TEXT [ INSTANCE , III ] := BLANK;
        "Blank print line"
    SIM_MON_INSTR := 5;
    E := S_JOB1_IO_TIME [ INSTANCE , S_JOB1_JOB_STEP
        [ INSTANCE ] ];
        "WAIT_TIME"
        INCR ( N );
    END;
16: BEGIN
    S_JOB1_CENTER ('JOB1 ENDS I/O',1,13,INSTANCE);
    SIM_MON_INSTR := 25;
    FOR III := 1 TO 130 DO
        K [ III ] := S_JOB1_TEXT [ INSTANCE , III ];
        "WRITE"
        INCR ( N );
    END;
17: BEGIN
    FOR III := 1 TO 130 DO
        S_JOB1_TEXT [ INSTANCE , III ] := BLANK;
        "Blank print line"
    SIM_MON_INSTR := 10;
    E := ENUM_TO_INTEGER ( S_DISK );
    F := S_JOB1_INDEX ;
    "RELEASE"
    INCR ( N );
    END;
18: BEGIN
    SIM_MON_INSTR := 16;
    E := 1;
    F := -1;
    "INC_COUNT"
    INCR ( N );
    END;

```

APPENDIX : SEQUENTIAL VERSION CODE

```
19: BEGIN
    S_JOB1_JOB_STEP [ INSTANCE ] :=
        (S_JOB1_JOB_STEP [ INSTANCE ] MOD
         S_JOB1_MAX_JOB_STEP ) + 1;
    S_JOB1_JOB_TIME [ INSTANCE ] := S_JOB1_EXEC_TIME [
        INSTANCE , S_JOB1_JOB_STEP [ INSTANCE ] ];
    SIM_MON_INSTR := 30;
    "NO OPERATION"
    INSTR_PTR [ N ] := 2;
END;
END "CASE"
END "JOB1TEST";
```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE JOB2TEST( N : INTEGER; VAR PARAM : SIM_INSTR_REC;
                   INSTANCE : INTEGER );
BEGIN
  WITH PARAM DO
    CASE INSTR_PTR [ N ] OF
      1: BEGIN
          SIM_MON_INSTR := 1;
          "ALIVE"
          INCR ( N );
        END;
      2: BEGIN
          SIM_MON_INSTR := 16;
          E := 2;
          F := 1;
          "INC_COUNT"
          INCR ( N );
        END;
      3: BEGIN
          SIM_MON_INSTR := 9;
          E := ENUM_TO_INTEGER ( S_CPU );
          F := 1;
          "SEIZE"
          INCR ( N );
        END;
      4: BEGIN
          S_JOB2_CENTER ( ' JOB2 STARTS SLICE AND JOB TIME IS'
                        ,10,34,INSTANCE);
          SIM_MON_INSTR := 26;
          E := S_JOB2_JOB_TIME [ INSTANCE ];
          "INT_CHAR"
          INCR ( N );
        END;
      5: BEGIN
          FOR III := 1 TO 6 DO
            S_JOB2_TEXT [ INSTANCE , 43 + III ] :=
              L [ III ];
          SIM_MON_INSTR := 25;
          FOR III := 1 TO 130 DO
            K [ III ] := S_JOB2_TEXT [ INSTANCE , III ];
            "WRITE"
            INCR ( N );
          END;
      6: BEGIN
          FOR III := 1 TO 130 DO
            S_JOB2_TEXT [ INSTANCE , III ] := BLANK;
            IF S_JOB2_JOB_TIME [ INSTANCE ] <= S_JOB2_SLICE
              THEN S_JOB2_DELTA [ INSTANCE ] :=
                S_JOB2_JOB_TIME [ INSTANCE ]
              ELSE S_JOB2_DELTA [ INSTANCE ] := S_JOB2_SLICE;
            SIM_MON_INSTR := 5;
            E := S_JOB2_DELTA [ INSTANCE ];
            "WAIT_TIME"
            INCR ( N );
          END;
    END CASE;
  END WITH;
END;

```

APPENDIX : SEQUENTIAL VERSION CODE

```

7: BEGIN
    "Calculate remaining job time"
    S_JOB2_JOB_TIME [ INSTANCE ] := S_JOB2_JOB_TIME
        [ INSTANCE ] - S_JOB2_DELTA [ INSTANCE ];
    SIM_MON_INSTR := 30;
    "NO OPERATION"
    INCR ( N );
END;
8: BEGIN
    SIM_MON_INSTR := 16;
    E := 2;
    F := -1;
    "INC_COUNT"
    INCR ( N );
END;
9: BEGIN
    S_JOB2_CENTER ( ' JOB2 ENDS SLICE AND JOBTIME IS '
        ,10,32,INSTANCE);
    SIM_MON_INSTR := 26;
    E := S_JOB2_JOB_TIME [ INSTANCE ];
    "INT_CHAR"
    INCR ( N );
END;
10: BEGIN
    FOR III := 1 TO 6 DO
        S_JOB2_TEXT [ INSTANCE , 43 + III ] :=
            L [ III ];
    SIM_MON_INSTR := 25;
    FOR III := 1 TO 130 DO
        K [ III ] := S_JOB2_TEXT [ INSTANCE , III ];
    "WRITE"
    INCR ( N );
END;
11: BEGIN
    FOR III := 1 TO 130 DO
        S_JOB2_TEXT [ INSTANCE , III ] := BLANK;
        "Blank out print line"
    IF S_JOB2_JOB_TIME [ INSTANCE ] <= 0
        "i. e. IF job step done "
        THEN INCR ( N ) "Proceed to I-0"
        ELSE INSTR_PTR [ N ] := 2; "Return to SEIZE"
    SIM_MON_INSTR := 10;
    E := ENUM_TO_INTEGER ( S_CPU );
    F := 1;
    "RELEASE"
END;
12: BEGIN
    SIM_MON_INSTR := 16;
    E := 1;
    F := 1;
    "INC_COUNT"
    INCR ( N );
END;

```

APPENDIX : SEQUENTIAL VERSION CODE

```

13: BEGIN
    SIM_MON_INSTR := 9;
    E := ENUM_TO_INTEGER ( S_DISK );
    F := S_JOB2_INDEX ;
    "SEIZE"
    INCR ( N );
END;
14: BEGIN
    S_JOB2_CENTER ( 'JOB2 STARTS I/O'
                    ,1,15,INSTANCE);
    SIM_MON_INSTR := 25;
    FOR III := 1 TO 130 DO
        K [ III ] := S_JOB2_TEXT [ INSTANCE , III ];
        "WRITE"
        INCR ( N );
    END;
15: BEGIN
    FOR III := 1 TO 130 DO
        S_JOB2_TEXT [ INSTANCE , III ] := BLANK;
        "Blank print line"
        SIM_MON_INSTR := 5;
        E := S_JOB2_IO_TIME [ INSTANCE ,
                             S_JOB2_JOB_STEP [ INSTANCE ] ];
        "WAIT_TIME"
        INCR ( N );
    END;
16: BEGIN
    S_JOB2_CENTER ( 'JOB2 ENDS I/O',1,13,INSTANCE);
    SIM_MON_INSTR := 25;
    FOR III := 1 TO 130 DO
        K [ III ] := S_JOB2_TEXT [ INSTANCE , III ];
        "WRITE"
        INCR ( N );
    END;
17: BEGIN
    FOR III := 1 TO 130 DO
        S_JOB2_TEXT [ INSTANCE , III ] := BLANK;
        "Blank print line"
        SIM_MON_INSTR := 10;
        E := ENUM_TO_INTEGER ( S_DISK );
        F := S_JOB2_INDEX ;
        "RELEASE"
        INCR ( N );
    END;
18: BEGIN
    SIM_MON_INSTR := 16;
    E := 1;
    F := -1;
    "INC_COUNT"
    INCR ( N );
END;

```

APPENDIX : SEQUENTIAL VERSION CODE

```
19: BEGIN
    S_JOB2_JOB_STEP [ INSTANCE ] :=
        (S_JOB2_JOB_STEP [ INSTANCE ] MOD
         S_JOB2_MAX_JOB_STEP ) + 1;
    S_JOB2_JOB_TIME [ INSTANCE ] := S_JOB2_EXEC_TIME [
        INSTANCE , S_JOB2_JOB_STEP [ INSTANCE ] ];
    SIM_MON_INSTR := 30;
    "NO OPERATION"
    INSTR_PTR [ N ] := 2;
END;
END "CASE"
END "JOB2TEST";
```

APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE MASTER_TIMER ( N : INTEGER; VAR PARAM : SIM_INSTR_REC);
BEGIN
  WITH PARAM DO
    CASE INSTR_FTR [ N ] OF
      1:BEGIN
        SIM_MON_INSTR := 1;
        "ALIVE"
        INCR ( N ) ;
      END;
      2: BEGIN
        S_MAST_FAC_SET [ 0 ] := 1; "CPU"
        S_MAST_FAC_SET [ 1 ] := 2; "DISKS"
        FOR III := 2 TO MAX_FACILITY DO
          S_MAST_FAC_SET [ III ] := 0;
        SIM_MON_INSTR := 8;
        E := 2;
        I := S_MAST_FAC_SET;
        INCR ( N );
        "DCL_FACILITIES"
      END;
      3: BEGIN
        SIM_MON_INSTR := 15;
        E := 2;
        "DCL_COUNTERS"
        INCR ( N );
      END;
      4:BEGIN
        SIM_MON_INSTR := 23;
        A := FALSE;
        B := FALSE;
        C := FALSE;
        "TRACE"
        INCR ( N ) ;
      END;
      5: BEGIN
        SIM_MON_INSTR := 5;
        E := 60000;
        "WAIT_TIME"
        INCR ( N );
      END;
      6: BEGIN
        SIM_MON_INSTR := 2;
        A := FALSE;
        "CANCEL"
        INCR ( N );
      END;
      7:BEGIN
        SIM_MON_INSTR := 20;
        D := TAP1;
        "REPORT_FACILITIES"
        INCR ( N );
      END;
    END CASE;
  END WITH;
END;

```

APPENDIX : SEQUENTIAL VERSION CODE

```
8:BEGIN
    SIM_MON_INSTR := 21;
    D := TAP1;
    "REPORT_COUNTERS"
    INCR ( N );
END;
9: BEGIN
    SIM_MON_INSTR := 22;
    D := TAP1;
    "REPORT_PROCESSES"
    INCR ( N );
END;
10: BEGIN
    SIM_MON_INSTR := 2;
    A := TRUE;
    "CANCEL"
    INCR ( N );
END;
END "CASE";
END "MASTER_TIMER";
```


APPENDIX : SEQUENTIAL VERSION CODE

```

PROCEDURE INITIALIZE;
  VAR
    NO , I : INTEGER;
BEGIN
  FOR NO := 1 TO 130 DO LG_TXT [ NO ] := ' ';

  "Initialize time list"

  FOR NO := 0 TO N_INTVLS DO
    WITH NOTICE [ NO + 1 ] DO
      BEGIN
        INTVL_PTR [ NO ] := NO + 1;
        IF NO = 0
          THEN PREV := N_INTVLS + 1
          ELSE PREV := (NO - 1) MOD (N_INTVLS + 1) + 1;
        NEXT := (NO + 1) MOD (N_INTVLS + 1) + 1;
        OCCURRENCE := CONV((NO + 1) * DT);
        DUMMY := TRUE;
        BEDTIME := 0.0;
      END "WITH";
    ICURRENT := 0;
    CURRENT := 1;
    LOWERBOUND := 0.0;

    "Initialize notice free list"

    FOR NO := N_INTVLS + 2 TO NOTICE_MAX DO
      WITH NOTICE [ NO ] DO
        BEGIN
          PREV := 0;
          NEXT := (NO + 1) MOD (NOTICE_MAX + 1);
          DUMMY := FALSE;
          BEDTIME := 0.0;
        END "OD";
      NFREE := N_INTVLS + 2;

      "Initialize events"

      FOR NO := 1 TO MAX_EVENT DO
        WITH EVENT [ NO ] DO
          BEGIN
            DELAYED := FALSE;
            Q_HEAD := 0;
            NUM_SIGNALS := 0.0;
            NUM_WAITS := 0.0;
            Q_AVE := 0.0;
            Q_MAX := 0;
            Q := 0;
            AVE_WAIT := 0.0;
          END "HTIW";
        END
      END
    END
  END

```

APPENDIX : SEQUENTIAL VERSION CODE

"Initialize processes' delay list"

```
FOR NO := MASTER_ID TO MAX_PROCESS DO
  WITH DELAY_LST [ NO ] DO
    BEGIN
      FOR M := 1 TO PMAX_EVENT DO
        WITH LINK [ M ] DO
          BEGIN
            PREV := 0;
            NEXT := 0;
            PREV_POS := 0;
            NEXT_POS := 0;
            EVENT_NO := 0;
          END "HTIW";
        WAIT_TIME := 0.0;
        ACTIVE_TIME := 0.0;
        REQ_TIME := 0.0;
        LST_TIME := 0.0;
        STATUS := DEAD;
      END "HTIW";
    END
  END
END
```

"Counters & facilities are initialized via user calls "

```
CDTN_LST := 0;
PROCS_DELAYED := 0;
ACTIVE_PROCS := MAX_PROCESS;
CLOCK := 0.0;
NO := 8667;
TRACE_FACILITY := FALSE;
TRACE_CNTR := FALSE;
TRACE_EVENT := FALSE;
SIM_STARTED := FALSE;
STALLED := 0;
FOR NO := 1 TO MAX_PROCESS DO TRACE_TABLE [ NO ] := PRINTR;
  HEAD := 1;
  LENGTH := 0;
  LIMIT := MAX_PROCESS;
  CANCELLED := FALSE;
  DELAY := FALSE;
  FOR NO := MAX_PROCESS DOWNT0 SCENARIO1 DO
    ACTIVE_PROCESSES [ ARRIVAL ] := NO;
  SCENARIO_NUM := MASTER_ID;
  FOR NO := MASTER_ID TO MAX_PROCESS DO
    INSTR_PTR [ NO ] := 1;
  END
END
```

APPENDIX : SEQUENTIAL VERSION CODE

"INSERT SCENARIO INITIALIZATION HERE "

"INITIALIZE JOB1TEST"

```
FOR NO := 1 TO 1 DO
  BEGIN
    FOR I := 1 TO 130 DO
      S_JOB1_TEXT [ NO , I ] := BLANK;
    S_JOB1_EXEC_TIME [ NO , 1 ] := 300;
    FOR I := 2 TO 7 DO
      S_JOB1_EXEC_TIME [ NO , I ] := 5;
    FOR I := 1 TO 7 DO
      S_JOB1_IO_TIME [ NO , I ] := 25;
    S_JOB1_JOB_STEP [ NO ] := 1;
    S_JOB1_JOB_TIME [ NO ] := S_JOB1_EXEC_TIME [ NO , 1 ];
  END;
```

"INITIALIZE JOB2TEST"

```
FOR NO := 1 TO 1 DO
  BEGIN
    FOR I := 1 TO 130 DO
      S_JOB2_TEXT [ NO , I ] := BLANK;
    S_JOB2_EXEC_TIME [ NO , 1 ] := 10;
    S_JOB2_IO_TIME [ NO , 1 ] := 25;
    S_JOB2_JOB_STEP [ NO ] := 1;
    S_JOB2_JOB_TIME [ NO ] := S_JOB2_EXEC_TIME [ NO , 1 ];
  END;
"%IF BREAK"
  BREAKPNT ( LINENUMBER ) ;
"%END BREAK"
END "INITIALIZE";
```

APPENDIX : SEQUENTIAL VERSION CODE

```
*****
* scenario calls simulation calls and scheduling *
*****
```

```
PROCEDURE GO;
  BEGIN
    REPEAT
      "%IF BREAK"
      BREAKPNT ( LINENUMBER ) ;
      "%END BREAK"
      CASE SCENARIO_NUM OF
        1:MASTER_TIMER (SCENARIO_NUM , PARAM);
        2:JOB2TEST ( SCENARIO_NUM , PARAM , 1);
        3:JOB1TEST ( SCENARIO_NUM , PARAM , 1);
        4:NULLPROC ( SCENARIO_NUM , PARAM );
        5:NULLPROC ( SCENARIO_NUM , PARAM );
        6:NULLPROC ( SCENARIO_NUM , PARAM );
        7:NULLPROC ( SCENARIO_NUM , PARAM );
        8:NULLPROC ( SCENARIO_NUM , PARAM );
        9:NULLPROC ( SCENARIO_NUM , PARAM );
        10:NULLPROC ( SCENARIO_NUM , PARAM );
      END;
      "%IF BREAK"
      BREAKPNT ( LINENUMBER );
      "%END BREAK"
    WITH PARAM DO
      CASE SIM_MON_INSTR OF
        1:"ALIVE"
          ALIVE;
        2:"CANCEL"
          CANCEL(A);
        3:"RENEW"
          RENEW;
        4:"CLOSE"
          CLOSE(D);
        5:"WAIT_TIME"
          WAIT_TIME( E );
        6:"WAIT_RAND"
          WAIT_RAND ( E , F );
        7:"TIME"
          H := TIME;
        8:"DCL_FACILITIES"
          DCL_FACILITIES ( E , I );
        9:"SEIZE"
          SEIZE ( E , F );
        10:"RELEASE"
          RELEASE ( E , F );
        11:"WAIT_EVENT"
          WAIT_EVENT ( E );
        12:"WAIT_VECTOR"
          WAIT_VECTOR ( E , J );
```

APPENDIX : SEQUENTIAL VERSION CODE

```

13:"SIGNAL"
    SIGNAL ( E );
14:"WAIT_UNTIL"
    WAIT_UNTIL ( E );
15:"DCL_COUNTERS"
    DCL_COUNTERS ( E );
16:"INC_COUNTERS"
    INC_COUNT ( E , F );
17:"ASSIGN_COUNT"
    ASSIGN_COUNT ( E , F );
18:"TEST"
    F := TEST ( E );
19:"UNIFORM"
    G := UNIFORM ( E , F );
20:"REPORT_FACS"
    REPORT_FACS ( D );
21:"REPORT_COUNTERS"
    REPORT_COUNTERS ( D );
22:"REPORT_PROCESSES"
    REPORT_PROCESSES ( D );
23:"TRACE"
    TRACE ( A , B , C );
24:"TRACE_FILE"
    TRACE_FILE ( D );
25:"WRITE"
    WRITE_SIM ( K );
26:"INT_CHAR"
    INT_CHAR ( E , L );
27:"REAL_CHAR"
    REAL_CHAR ( H , E , F , M );
28:"DISPLAY_TIME"
    DISPLAY_TIME;
29:"REPORT_EVENTS"
    REPORT_EVENTS ( D );
30:"NO OPERATION";
    END"CASE" "WITH";
"%IF BREAK"
BREAKPNT ( LINENUMBER );
"%END BREAK"

```

APPENDIX : SEQUENTIAL VERSION CODE

" Check if process has DELAYed itself"
 " Scheduling code "

```

    IF DELAY
      THEN
        BEGIN
          "%IF BREAK"
          BREAKPNT ( LINENUMBER );
          "%END BREAK"
          DELAY := FALSE;
          IF NOT EMPTY
            THEN
              BEGIN
                "%IF BREAK"
                BREAKPNT ( LINENUMBER );
                "%END BREAK"
                "There is an active process in the queue"
                SCENARIO_NUM := ACTIVE_PROCESSES
                  [ DEPARTURE ];
              END
            ELSE
              BEGIN
                "%IF BREAK"
                BREAKPNT ( LINENUMBER );
                "%END BREAK"
                "Find next process to activate"
                "%IF BUG"
                DMP_NOTICE ( CURRENT);
                "%END BUG"
                TIME_MOVES_ON;
                "%IF BUG"
                DMP_NOTICE ( CURRENT );
                "%END BUG"
                SCENARIO_NUM:=NOTICE [ CURRENT ] .SLEEP;
                UPDATE_TIME ;
                "%IF BUG"
                DMP_NOTICE ( CURRENT );
                "%END BUG"
              END;
            END;
          UNTIL ( CANCELLED );
        END "GO";

BEGIN
  INITIALIZE;
  GO;
END.
```

COMPARISON OF THE EFFECTS OF
CODING TECHNIQUES
ON SIMULATION CONCEPTS
IN PASCAL

by

BRIAN JOHN FERGUSON

B.S., University of California, Davis 1974

ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1980

ABSTRACT

This report examines the utility and performance of extending the Pascal language to include simulation mechanisms. Two different approaches are examined. The approaches examined include the extension of the Concurrent Pascal language and the Sequential Pascal language to include simulation mechanisms. Method one was written in Concurrent Pascal and used the concepts of Per Brinch Hansen's Monitor, Class, and Process. Method two was written in Sequential Pascal. The two approaches are compared against each other to determine the utility, performance and other factors which would influence the decision to use one approach rather than the other. Each method was implemented and tested by running actual simulations. These results are used in comparing and contrasting the two methods. The results of these two methods indicate that the extension of Sequential Pascal has a faster execution speed compared with the Concurrent Pascal method. The Concurrent Pascal method is better defined, easier to use, and has a better implementation structure than the Sequential Pascal method.