Proceedings of the 1984
Winter Simulation Conference
S. Sheppard, U. Pooch, D. Pegden (eds.)

IMPLEMENTATION OF OPERATIONAL EVALUATION MODELING IN PASCAL

John A. Quandt
General Dynamics, Pomona Division
Pomona, Ca 91766

ABSTRACT

This paper presents the results of a study of the implementation of Operational Evaluation Modeling (OEM) in the structured programming language Pascal. The study compares the FORTRAN and Pascal programming languages, and then discusses implementation of OEM simulation in Pascal. The results show it is possible to develop OEM simulations in Pascal using a top-down approach that parallels the development of OEM models themselves, and that the user benefits from this approach in the clarity of the resultant Pascal program.

OPERATIONAL EVALUATION MODELING

Operational Evaluation Modeling (OEM) has been described as "being comprised of a collections of ideas, concepts, and techniques which provide a means to evaluate a system from the user's point of view". "The main concepts incorporated involve parallel processes, discrete event simulations, and queuing theory"[1]. The system user begins by defining a crude version of the entire system he wishes to model. The model is then run varying the inputs to determine the areas of the model which are sensitive to change. These areas are then refined by the user to obtain a model which accurately simulates the system's response. The main thrust of OEM is to quickly develop system models from a top-down approach and only refine the overall system model where required.

DEVELOPMENT OF PROGRAMMING LANGUAGES

When computers were first developed, no high level programming languages existed. All programs were entered by setting switches on the control panel and pressing the enter button [2]. Needless to say, this was not conducive to writing and entering large programs.

Next to be developed was assembly language, which is a symbolic representation of the processor's machine language. While easier than machine language coding, assembly language programming is both time-consuming and difficult to read. Development of assembly language programs are typically conducted from the bottom up, coding numerous individual assembly language modules and then linking them together into one large program. Assembly language programming is also not easily transportable from one machine to another, as different processors have different architectures and instruction sets.

Next in 1956 came the first generally-accepted high level language, FORTRAN, which was designed primarily for scientific use. Unfortunately, FORTRAN was structured much the same as the assembly language programming it replaced. This becomes evident once one examines the action resulting from a logical comparison. As in assembly language, FORTRAN logical IF statements result in immediate branching to another location in the program, if the conditions for the IF statement are met. While easier for the compiler author to implement because the high-level language so closely resembles the assembly language structure he has to work with, these bottom-up design concepts make it difficult for the high-level language software author to follow the logical flow throughout his program. Recalling the period in which FORTRAN was developed, it is easy to see why many of the bottom-up design concepts of assembly language were incorporated.

Gradually, the computer science field has moved towards a top-down design structure concept that is usually referred to as structured programming [3]. In 1968, Niklaus Wirth defined a language, called Pascal, named after the French mathematician [2], to teach structured programming concepts.

PASCAL VERSUS FORTRAN

The primary advantages of using a structured high level language, like Pascal, over FORTRAN are clarity and flexibility. Also note the top-down structure of Pascal programs more closely mimics the development of OEM models than does FORTRAN. Both Pascal and OEM start with a loose top-down design that is expanded upon by piece-wise refinement.

Pascal was designed to be clear, readable, and unambiguous [4]. There are many reasons for the added clarity being inherent in the design of Pascal programs. Top-down design is more natural for people, who tend to solve problems by defining a general approach to the problem and then refining it until the problem is solved [3]. Breaking parts of the program into small modules, each with its own specific task to perform, is easier to comprehend than having one large cumbersome program.

Pascal also requires strong data typing. All variables must be clearly defined before they can be used. There are no default variable types. The majority of variables used in the program are defined in the main declaration section, and, as such, are available globally without the confusing and repetitious chore of defining them in every routine's COMMON block, as in FORTRAN. For example, in the dice program of appendix 1, "FirstRoll" and "NextRoll" are declared to be variables of type "INTEGER" in the main program declaration (lines 8 and 9), and as such are available globally throughout the program, while "Toss" is declared to be of type "INTEGER" in the "RollEm" procedure (line 11), and as such is defined only within that procedure (lines 11 through 15). Another example of the strong typing in Pascal is the distinction made between the value assignment operator

(:=) and the equality test operator (=). Improper use of these operators results in the declaration of a syntax error upon compilation. Also, there is no need to limit the variable and subroutine names in Pascal to six letters, as in FORTRAN. The name can be as long as necessary to be descriptive of the function it performs. For example, it's far easier to understand "PopEventQ" than the FORTRAN OEM equivalent [1], which was "REMOVE(I,1)". In the particular Pascal implementation used in the preparation of this paper, Apple Pascal 1.1 [5], both upper and lower case letters were available to add clarity to the user-defined names. For example, "FirstRoll" is easier to read than "FIRSTROLL".

Pascal also encourages clarity by allowing indentation to show the limits of effect for any logical block of code. Most FORTRAN and BASIC language implementations consider indentation an error and either remove it or declare it a syntactical error. Looking at the dice program in appendix 1, one observes that statements at the same lexical level are indented to the same column. For every "BEGIN", there is a corresponding "END" at the same column to show the limit of effect of this block of code. The "BEGIN/END" pair farthest to the left (lines 29 and 42) defines the limits of the main program. The indentation is merely a user convenience, as Pascal compilers require the semi-colon or other delimiter to define the break between statements and ignores blanks.

One other change to note is that standard Pascal prohibits the use of the "GOTO" statement. "GOTO's" were found to be disruptive in terms of the user's ability to follow the natural and logical flow of the program. Instead, more logical statements were added to provide for conditional execution, such as "IF..THEN..ELSE", "CASE", "REPEAT..UNTIL", and "WHILE". Once again, examining the dice program in appendix 1, one notes the "REPEAT..UNTIL" statements. These define the limits for conditional execution of the code present between corresponding limits, i.e. all the code between the "REPEAT" (line 30) and the "UNTIL HellFreezesOver" (line 41) will be executed until the parameter "HellFreezesOver" becomes "TRUE", which will literally take "until hell freezes over", as the value of a constant cannot be modified. The other new conditional statement utilized in the dice program is the "CASE" statement (line 35). This is used to replace a series of "IF..THEN..ELSE" statements.

One other feature that FORTRAN does not have is recursion, i.e. a FORTRAN subroutine cannot call itself, whereas a Pascal procedure can. Sometimes it is more straightforward to describe a task recursively than by other means. For example, this can be seen in appendix 1, where the dice game contains the recursive procedure "RollAgain". This procedure will call itself until such time as the roll of the dice equals either the first roll or seven. This form more closely approximates the mental processes utilized by a person to perform this action. For many applications, the recursive description of a process is a "perfect example of top-down programming" [7].

Pascal also tends to be a more flexible language than FORTRAN. In Pascal, the programmer has the option to define his own data types, even associating dissimilar data types. For example, the implementation of the queuing data structure for the program in appendix 3 directly associates a real number, three integer numbers, and a pointer. The declaration of this structure is performed in the "TYPE" definition.

Instead of using the confusing approach of numeric subscripts to access each field within the defined record of "QueueEntry", Pascal uses the more straightforward approach of addressing each field by its user-defined name. For example, the field "Time" would be addressed as follows:

```
TimeNow := QueueEntry.Time;
```

This statement says that the variable "TimeNow" is assigned the value of the "Time" field within the record of "QueueEntry". Also associated with the real number value of time are fields to represent event numbers, process numbers, duplicate process numbers, and a pointer to the next entry in the linked list which comprises the queue. FORTRAN has no ability to provide such a direct association or means of addressing.

By now you're probably asking yourself, "What's a Pointer?". Pointers can be used in Pascal to construct dynamically allocated data structures, such as queues. The pointer is represented symbolically by "^", which reads as "points to". Pointers correspond to an address in memory and, as such, provide a means to link records together to generate a structure such as a queue. A list would be constructed by defining a pointer to be the start of the list. The list is terminated with a pointer value of NIL, which can be tested for when searching through a list. A graphical representation of a linked list is presented in figure 1. Insertion of an element into a linked list is accomplished without physically moving any data by altering the pointer used to address the next record in the list. Deletion is accomplished, just as easily, by altering pointers to eliminate the record from the list. Thus the event scheduling and wait queues of OEM can be implemented dynamically with logically associated elements rather than with a confusing collection of disassociated fixed length arrays.

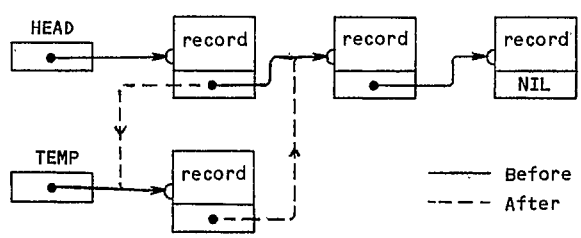

Figure 1: Insertion of a Record into a Linked List

As a result of these conceptual changes, well-written Pascal programs tend to be self-documenting [6]. One criticism of this is that well-written Pascal programs also tend to be wordy. However, it is this wordiness which tends to make Pascal programs easier to read, much like reading a recipe. One may verify this by comparing the Pascal version of the dice game in appendix 1 with the FORTRAN version in appendix 2. Both programs execute almost identically from the user's point of view.

THE CALCULATOR FACTORY SCENARIO

The system simulated was a standard classical parallel process, the producer-consumer process of a calculator factory [1]. The system simulates the actions of a calculator assembler, the producer, placing his completed calculators on a conveyor belt, from which a packer, the consumer, removes the calculators and packs them for shipment. Figure 2 shows how a conflict arises over access to the conveyor belt, as a door exists on each end of the belt, as a safety measure, with an interlocking mechanism to prevent having both doors open at the same time. This system is represented graphically by the directed graph model of figure 3.

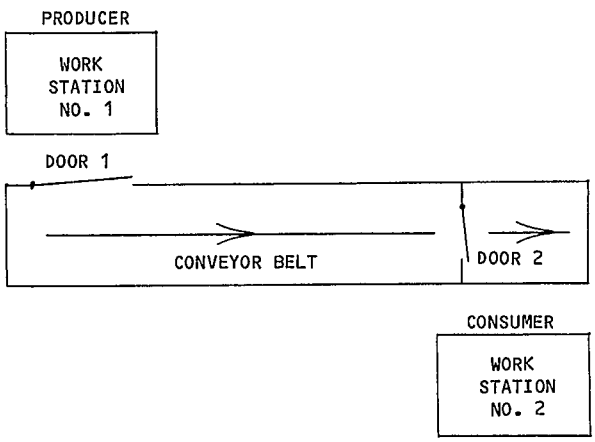

Figure 2: The Calculator Factory Scenario

OEM simulates this parallel process through the use of two scheduling queues. The first list is an event scheduling queue, which schedules events to occur on a time-ordered basis, dependent on the length of time it takes to complete each step of the process. The second list is a wait queue, which schedules events to occur, dependent on the availability of limited, shared resources. The wait queue is checked on a first-come, first-served basis. More information on this subject is available in reference [1].

Based on the directed graph model, the producer-consumer system behaves as follows:

1. The system begins operation initializing the number of units on the belt as zero, setting the number of empty positions on the belt to the default buffer size, and indicating the belt as being free. Initialization proceeds to schedule the "Calculator Completed" event (E2) and places the "Calculator Available" event in the wait list, i.e. the packer will wait until a calculator is available.
2. When event E2 occurs, the action of event E2 places event E3, the "Space On Belt" event, on the wait list.
3. Upon checking the wait list, if space exists on the belt, then event E3 occurs and event E4, the "Belt Free" event, is placed on the wait list.
4. Upon checking the wait list again, if the belt is available, then event E4 occurs and schedules event E5, the "Calculator On Belt" event, on the event list.
5. The decision is then made on whether to terminate the assembly operation for the day, or not. Based on that decision, the assembler process either proceeds to the "PRD" state and begins assembling another calculator or proceeds to the "I1" state and waits for the packer before ending the day's operation.

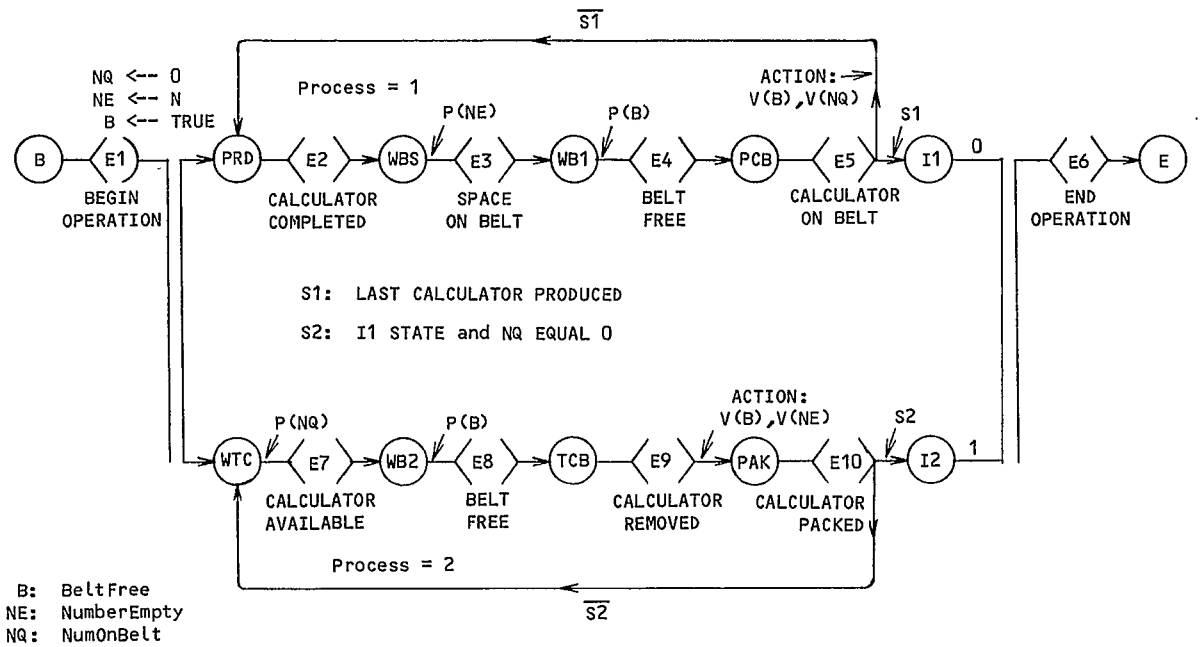

Figure 3: Directed Graph Model of the Calculator Factory

6. Meanwhile, the packer has been waiting in the "WTC" state for a calculator to appear on the conveyor belt. As soon as one does, event E7, the "Calculator Available" event, occurs and event E8, the "Belt Free" event, is placed in the wait list.
7. When the conveyor belt becomes available, event E8 occurs and event E9, the "Calculator Removed" event, is scheduled on the event list.
8. When event E9 occurs in time, event E10, the "Calculator Packed" event is scheduled in the event list.
9. Here the packer has to decide whether to proceed back to the "WTC" state, or continue on to the "I2" state, signifying the belt is empty and the assembler has finished for the day. At this point, if the end of the day has been reached, event E6, the "End Operation" event is scheduled.

Implementation in Pascal

The producer-consumer problem described herein has previously been implemented in FORTRAN [1, App. B]. As previously mentioned, FORTRAN is not ideally suited to development of these simulations. The Pascal version of this system incorporates three simple queues to replace FORTRAN's fourteen fixed-length arrays and seven pointer variables. Discussion of the specifics of the Pascal implementation of the Calculator Factory Scenario follows.

One implementation of the producer-consumer process in Pascal is given in appendix 3. Examining the program listing, one notes the following features:

1. The program begins with a declaration block. Here all external software libraries, global constants, global data type definitions, and global variables are declared. Pascal requires all utilities, constants, user-defined data types, and variables be defined prior to their usage. The type declaration contains the definition of the entries in the queues, conveniently addressed as a "QueueEntry" with fields of "Time", "Event", "Process", "DupProcess", and "NextPointer". Note that "NextPointer" is declared to be of type "QPointer", which is defined as being equal to "^QueueEntry". What this means is that "NextPointer" is an address which points to a record of "QueueEntry". That is, variables of type "QPointer" are used to point to records of the five field long "QueueEntry". As one of the fields of "QueueEntry" is a "QPointer", then one can link records of "QueueEntry" together by having each record point to the next record in the list to create a linked list, i.e. a queue.
2. Definition of each of the global variables is fairly self-explanatory due to Pascal's ability to handle names with more than six letters and the ability to use both upper and lower case letters.
3. Function "Rand" generates a pseudo-random series of real numbers between zero and one.
4. Function "Gamma" generates a random number with a range of values and distribution controlled by the random variables, RndVar, used.
5. Procedure "RndVarInit" is used to initialize the random variables, RndVar.
6. Procedure "NameInit" is used to initialize the event, producer state, and consumer state names.
7. Procedure "ProbInit" is used to initialize the value of the idle probability.
8. Procedure "QInit" is used to initialize the event, wait, and spare record queues.
9. Procedures "PopSpareQ" and "PushSpareQ" are used to remove and place records onto the spare record queue. While capable of generating over twenty-three hundred records of "QueueEntry" for this problem, Apple Pascal 1.1 [5] lacks the "DISPOSE" intrinsic, which would allow automatic reallocation of unused dynamically allocated memory, commonly referred to as garbage collection. Thus, in order to conserve memory, queueing records not in use are placed in a spare record queue. Only when the spare record queue is empty are new records created.
10. Procedure "PopEventQ" removes the first record from the event queue, i.e. the record with the lowest "Time".
11. Procedure "FileEvent" is used to place a record on the event queue in a time-ordered manner, from lowest to highest.
12. Procedure "InsertEvent" is called by "FileEvent" to search through the event queue and place the new event in the proper place. "InsertEvent" calls itself recursively until it finds the proper place.
13. Procedure "RemoveWait" is used in the process of removing records from the wait queue. The handling of the pointers is managed in the "CheckWaitQ" procedure.
14. Procedure "FileWait" is used to place a new record at the end of the wait queue.
15. Procedure "AddWait" is called by "FileWait" to search through the wait queue until it finds the end of the queue.
16. Procedure "UnfileAll" is used to release all dynamically allocated memory used by the queues during the day's operation.
17. Procedure "TimeTranslate" is used to convert the value of time from a real number of seconds to integer hours, integer minutes, and real seconds.
18. Procedure "EventTrace" is used to print out the event trace data.
19. Procedures "Event2" through "Event10", "ProdSimulation", "ConsSimulation", and "Events" are used to handle the processing of events.
20. Procedure "StatsCollect" is used to collect data on the wait times for each wait state and the measure of effectiveness.
21. Procedure "CheckWaitQ" is used to scan through the wait queue to determine if any wait events can be performed.
22. Procedure "StateTrace" is used to print out the state trace data.
23. Procedure "Initialize" is used to initialize the simulation at the start of each operation, i.e. day.
24. Procedures "VarUpdate", "RndVarUpdate", and "ProbUpdate" are used to optionally change the default values of the random variables and idle probability.
25. Procedure "ReadInputs" is used to read the user selected program options.
26. Procedure "Report" is used to print out a report at the end of the simulation run.
27. The block of code between the last BEGIN/END is the main program. The main program starts by initializing the random variables, idle probability, event and state names, and the event, wait, and spare record queues. The program then proceeds to read user inputs. The main processing loops follow. The outer "WHILE" loop executes until such time as the user inputs from "ReadInputs" tell the program to stop. The next inner loop, the "FOR" loop, executes until the total number of days have been processed. The inner "REPEAT..UNTIL" loop executes until "Event6" clears the event and wait queues at the end of the day.

In addition to resulting in a program that was far easier to understand, the Pascal version executed faster than the FORTRAN. This was expected, as Pascal programs tend to execute "50% faster than FORTRAN" versions of the same program on the same machine [6]. My own personal experience on the Apple [5] shows that similar programs tend to run 25% faster in Pascal than FORTRAN, although this is with similar structures between both programs. Restructuring the Pascal version utilizing the additional features of Pascal does tend to improve upon this.

CONCLUSIONS

This paper has presented the results of a study into the implementation of Operational Evaluation Modeling (OEM) in Pascal, a highly-structured, top-down design programming language. This is significant as the simulation community is just becoming aware of the new structured programming languages, such as Pascal and Ada. As a result, this paper represents one of the first implementations of OEM in a top-down structured language. Pascal was demonstrated to be superior to FORTRAN for this application, due to the similarity in structure between the OEM simulation technique and the top-down program development technique used in the Pascal programming language, resulting in a program that was far easier to read and comprehend than the FORTRAN version. A side benefit of programming highly-structured programs in Pascal is slightly faster execution times than the comparable program written in FORTRAN.

APPENDIX 1: PASCAL LISTING OF DICE GAME PROGRAM

```
line
  1  PROGRAM Craps;
  2
  3    USES APPLESTUFF;
  4
  5    CONST
  6      HellFreezesOver = FALSE;
  7
  8    VAR
  9      FirstRoll, NextRoll : INTEGER;
 10
 11    PROCEDURE RollEm (VAR Toss : INTEGER);
 12      BEGIN
 13        Toss := RANDOM MOD 6 + 1;
 14        Toss := Toss + RANDOM MOD 6 + 1;
 15      END;
 16
 17    PROCEDURE RollAgain;
 18      BEGIN
 19        RollEm (NextRoll);
 20        WRITELN (' NEXT ROLL IS ', NextRoll:2);
 21
 22        IF NextRoll = FirstRoll THEN
 23          WRITELN (' YOU WIN');
 24        ELSE IF NextRoll = 7 THEN
 25          WRITELN ( 'YOU LOSE');
 26        ELSE RollAgain;
 27      END;
 28
 29  BEGIN
 30    REPEAT
 31      RollEm (FirstRoll);
 32      WRITELN;
 33      WRITELN (' YOU ROLLED A ', FirstRoll:2);
 34
 35      CASE FirstRoll OF
 36        7, 11            : WRITELN (' YOU WIN');
 37        2, 3, 12         : WRITELN (' YOU LOSE');
 38        4, 5, 6, 8, 9, 10 : RollAgain;
 39        END;
 40
 41    UNTIL HellFreezesOver;
 42  END.
```

APPENDIX 2: FORTRAN LISTING OF DICE GAME PROGRAM

```
line
  1  $USES APPLESTUFF
  2        PROGRAM CRAPS
  3      1 FORMAT(/' YOU ROLLED A ',I2)
  4      2 FORMAT(' YOU WIN')
  5      3 FORMAT(' YOU LOSE')
  6      4 FORMAT(' NEXT ROLL IS ',I2)
  7    100 CALL ROLLEM(IROLL1)
  8        WRITE(*,1) IROLL1
  9        IF(IROLL1.NE.7.AND.IROLL1.NE.11) GOTO 200
 10    150 WRITE(*,2)
 11        GOTO 100
 12    200 IF(IROLL1.NE.2.AND.IROLL1.NE.3.AND.
          1   IROLL1.NE.12) GOTO 300
 13    250 WRITE(*,3)
 14        GOTO 100
 15    300 CALL ROLLEM(IROLL2)
 16        WRITE(*,4) IROLL2
 17        IF(IROLL2.EQ.IROLL1) GOTO 150
 18        IF(IROLL2.EQ.7) GOTO 250
 19        GOTO 300
 20        END
 21        SUBROUTINE ROLLEM(ITOSS)
 22        ITOSS=MOD(RANDOM(),6)+1
 23        ITOSS=ITOSS+MOD(RANDOM(),6)+1
 24        RETURN
 25        END
```

APPENDIX 3: THE CALCULATOR FACTORY PROGRAM

```
PROGRAM ProducerConsumer;

USES APPLESTUFF, TRANSCEND;

CONST
  BufferSize = 10;
  IdleDefault = 0.15;
  TimeInitial = 0.0;

TYPE
  QPointer = ^QueueEntry;
  QueueEntry = RECORD
                 Time        : REAL;
                 Event       : INTEGER;
                 Process     : INTEGER;
                 DupProcess  : INTEGER;
                 NextPointer : QPointer
               END;

VAR
  Process, EventNumber, NumberUnits,
  NumberEmpty, DupProcess, NumberMissions,
  TotalMissions, NumOnBelt, I, J, ProdState,
  ConsState, Hours, Minutes : INTEGER;

  NumberWait : ARRAY [1..4] OF INTEGER;

  Time, TimeNow, Temp, AveMOE, AveThroughput,
  ProbIdle, Seconds : REAL;

  RndVar : ARRAY [1..5, 1..4] OF REAL;
  TimeWait, AveWait : ARRAY [1..4] OF REAL;
```

```
  Stop, StraceFlag, EtraceFlag, BeltFree,
  RemoveFlag : BOOLEAN;

  Key : CHAR;

  EventName : ARRAY [1..10] OF STRING;
  ConsName, ProdName : ARRAY [1..6] OF STRING;
  WaitName : ARRAY [1..4] OF STRING;

  HeadEventQ, HeadWaitQ, LastWaitPointer,
  ThisWaitPointer, HeadSpareQ, TempPointer : QPointer;
  Heap : ^INTEGER;

FUNCTION Rand : REAL;
  BEGIN
    Rand := RANDOM MOD 1024 / 1023.0;
  END;

FUNCTION Gamma (Select : INTEGER) : REAL;
  VAR
    DegOfFreedom : INTEGER;
    Temp : REAL;
  BEGIN
    IF RndVar [1, Select] < Rand THEN
      Gamma := 1.0E10
    ELSE
      BEGIN
        DegOfFreedom := TRUNC (RndVar [2, Select]);
        Temp := 1.0;

        FOR I := 1 TO DegOfFreedom DO
          Temp := Temp * Rand;

        IF Temp = 0.0 THEN Temp := 1.0E-15;
        Temp := -1.0 * RndVar [3, Select] * LN (Temp)
                / DegOfFreedom;
        IF Temp < RndVar [4, Select] THEN Temp :=
           RndVar [4, Select];
        IF Temp > RndVar [5, Select] THEN Temp :=
           RndVar [5, Select];
        Gamma := Temp;
      END;
  END;

PROCEDURE RndVarInit;
  BEGIN
    RndVar [1, 1] := 1.0; RndVar [2, 1] := 3.0;
    RndVar [3, 1] := 60.0; RndVar [4, 1] := 10.0;
    RndVar [5, 1] := 150.0;
    RndVar [1, 2] := 1.0; RndVar [2, 2] := 10.0;
    RndVar [3, 2] := 10.0; RndVar [4, 2] := 0.0;
    RndVar [5, 2] := 30.0;
    RndVar [1, 3] := 1.0; RndVar [2, 3] := 10.0;
    RndVar [3, 3] := 10.0; RndVar [4, 3] := 0.0;
    RndVar [5, 3] := 30.0;
    RndVar [1, 4] := 1.0; RndVar [2, 4] := 2.0;
    RndVar [3, 4] := 90.0; RndVar [4, 4] := 10.0;
    RndVar [5, 4] := 180.0;
  END;

PROCEDURE NameInit;
  BEGIN
    EventName [ 1] := 'BEGIN OPERATION     ';
    EventName [ 2] := 'CALCULATOR COMPLETED';
    EventName [ 3] := 'SPACE ON BELT       ';
    EventName [ 4] := 'BELT FREE           ';
    EventName [ 5] := 'CALCULATOR ON BELT  ';
    EventName [ 6] := 'END OPERATION       ';
    EventName [ 7] := 'CALCULATOR AVAILABLE';
    EventName [ 8] := 'BELT FREE           ';
    EventName [ 9] := 'CALCULATOR REMOVED  ';
    EventName [10] := 'CALCULATOR PACKED   ';

    ConsName [1] := '    ';
    ConsName [2] := 'WTC ';
    ConsName [3] := 'WB2 ';
    ConsName [4] := 'TCB ';
    ConsName [5] := 'PAK ';
    ConsName [6] := 'I2  ';

    ProdName [1] := '    ';
    ProdName [2] := 'PRD ';
    ProdName [3] := 'WBS ';
    ProdName [4] := 'WB1 ';
    ProdName [5] := 'PCB ';
    ProdName [6] := 'I1  ';

    WaitName [1] := 'WBS';
    WaitName [2] := 'WB1';
    WaitName [3] := 'WTC';
    WaitName [4] := 'WB2';
  END;

PROCEDURE ProbInit;
  BEGIN
    ProbIdle := IdleDefault;
  END;

PROCEDURE QInit;
  BEGIN
    MARK (Heap);
    HeadEventQ := NIL;
    HeadWaitQ  := NIL;
    HeadSpareQ := NIL;
  END;

PROCEDURE PopSpareQ (VAR SparePointer : QPointer);
  BEGIN
    SparePointer := HeadSpareQ;
    HeadSpareQ   := HeadSpareQ^.NextPointer;
  END;

PROCEDURE PushSpareQ (VAR SparePointer : QPointer);
  BEGIN
    SparePointer^.NextPointer := HeadSpareQ;
    HeadSpareQ := SparePointer;
  END;

PROCEDURE PopEventQ;
  BEGIN
    TimeNow     := HeadEventQ^.Time;
    EventNumber := HeadEventQ^.Event;
    Process     := HeadEventQ^.Process;
    DupProcess  := HeadEventQ^.DupProcess;
    TempPointer := HeadEventQ;
    HeadEventQ  := HeadEventQ^.NextPointer;
    PushSpareQ (TempPointer);
  END;

PROCEDURE FileEvent;
  VAR
    NewEventPointer : QPointer;

  PROCEDURE InsertEvent (VAR LastEventPointer :
                         QPointer);
    BEGIN
      IF LastEventPointer = NIL THEN
        BEGIN
          NewEventPointer^.NextPointer := NIL;
          LastEventPointer := NewEventPointer;
        END
      ELSE IF NewEventPointer^.Time <
              LastEventPointer^.Time THEN
        BEGIN
          NewEventPointer^.NextPointer :=
            LastEventPointer;
          LastEventPointer := NewEventPointer;
        END
      ELSE
        InsertEvent (LastEventPointer^.NextPointer);
```

```
    END;

  BEGIN
    IF Time < 1.0E9 THEN
      BEGIN
        IF HeadSpareQ = NIL THEN
          NEW (NewEventPointer)
        ELSE
          PopSpareQ (NewEventPointer);
          NewEventPointer^.Time       := Time;
          NewEventPointer^.Event      := EventNumber;
          NewEventPointer^.Process    := Process;
          NewEventPointer^.DupProcess := DupProcess;
          InsertEvent (HeadEventQ);
      END;
  END;

PROCEDURE RemoveWait;
  BEGIN
    Time       := ThisWaitPointer^.Time;
    DupProcess := ThisWaitPointer^.DupProcess;
    RemoveFlag := TRUE;
  END;

PROCEDURE FileWait;
  VAR
    NewWaitPointer : QPointer;

  PROCEDURE AddWait (VAR WaitPointer : QPointer);
    BEGIN
      IF WaitPointer = NIL THEN
        WaitPointer := NewWaitPointer
      ELSE
        AddWait (WaitPointer^.NextPointer);
    END;

  BEGIN
    IF HeadSpareQ = NIL THEN
      NEW (NewWaitPointer)
    ELSE
      PopSpareQ (NewWaitPointer);

    NewWaitPointer^.Time        := Time;
    NewWaitPointer^.Event       := EventNumber;
    NewWaitPointer^.Process     := Process;
    NewWaitPointer^.DupProcess  := DupProcess;
    NewWaitPointer^.NextPointer := NIL;
    AddWait (HeadWaitQ);
  END;

PROCEDURE UnfileAll;
  BEGIN
    RELEASE (Heap);
  END;

PROCEDURE TimeTranslate;
  BEGIN
    Temp := TimeNow / 3600;
    Hours := TRUNC (Temp);
    Temp := (Temp - Hours) * 60;
    Minutes := TRUNC (Temp);
    Seconds := (Temp - Minutes) * 60;
  END;

PROCEDURE EventTrace;
  BEGIN
    TimeTranslate;
    WRITELN (Hours:2, ':', Minutes:2, ':',
        Seconds:6:2, '   ', EventName [EventNumber],
        '  EventNumber= ', EventNumber:3,
        '   Process = ', Process:3);
  END;

PROCEDURE Event2;
  BEGIN
    IF EtraceFlag THEN EventTrace;
    NumberUnits := NumberUnits + 1;
    ProdState := 2;
    Time := TimeNow;
    EventNumber := 3;
    FileWait;
  END;

PROCEDURE Event3;
  BEGIN
    IF NumberEmpty <> 0 THEN
      BEGIN
        RemoveWait;
        IF EtraceFlag THEN EventTrace;
        TimeWait [1] := TimeWait [1] +
          (TimeNow - Time);
        NumberWait [1] := NumberWait [1] + 1;
        NumberEmpty := NumberEmpty - 1;
        ProdState := 3;
        Time := TimeNow;
        EventNumber := 4;
        FileWait;
      END;
  END;

PROCEDURE Event4;
  BEGIN
    IF BeltFree THEN
      BEGIN
        RemoveWait;
        IF EtraceFlag THEN EventTrace;
        TimeWait [2] := TimeWait [2] +
          (TimeNow - Time);
        NumberWait [2] := NumberWait [2] + 1;
        BeltFree := FALSE;
        ProdState := 4;
        Time := TimeNow + Gamma (2) *
          (BufferSize - NumOnBelt);
        EventNumber := 5;
        FileEvent;
      END;
  END;

PROCEDURE Event5;
  BEGIN
    IF EtraceFlag THEN EventTrace;
    NumOnBelt := NumOnBelt + 1;
    BeltFree := TRUE;

    IF ProbIdle >= Rand THEN
      ProdState := 5
    ELSE
      BEGIN
        ProdState := 1;
        Time := TimeNow + Gamma (1);
        EventNumber := 2;
        FileEvent;
      END;
  END;

PROCEDURE Event6;
  BEGIN
    IF EtraceFlag THEN EventTrace;
    ProdState := 0;
    ConsState := 0;
    HeadEventQ := NIL;
    HeadWaitQ  := NIL;
  END;
```

```
PROCEDURE ProdSimulation;
  BEGIN
    CASE EventNumber OF
      2 : Event2;
      3 : Event3;
      4 : Event4;
      5 : Event5;
      6 : Event6;
      END;
  END;

PROCEDURE Event7;
  BEGIN
    IF NumOnBelt <> 0 THEN
      BEGIN
        NumOnBelt := NumOnBelt - 1;
        RemoveWait;
        IF EtraceFlag THEN EventTrace;
        TimeWait [3] := TimeWait [3] +
          (TimeNow - Time);
        NumberWait [3] := NumberWait [3] + 1;
        ConsState := 2;
        Time := TimeNow;
        EventNumber := 8;
        FileWait;
      END;
  END;

PROCEDURE Event8;
  BEGIN
    IF BeltFree THEN
      BEGIN
        BeltFree := FALSE;
        RemoveWait;
        IF EtraceFlag THEN EventTrace;
        TimeWait [4] := TimeWait [4] +
          (TimeNow - Time);
        NumberWait [4] := NumberWait [4] + 1;
        ConsState := 3;
        Time := TimeNow + Gamma (3);
        EventNumber := 9;
        FileEvent;
      END;
  END;

PROCEDURE Event9;
  BEGIN
    IF EtraceFlag THEN EventTrace;
    NumberEmpty := NumberEmpty + 1;
    BeltFree := TRUE;
    ConsState := 4;
    Time := TimeNow + Gamma (4);
    EventNumber := 10;
    FileEvent;
  END;

PROCEDURE Event10;
  BEGIN
    IF EtraceFlag THEN EventTrace;
    IF (ProdState = 5) AND (NumOnBelt = 0) THEN
      BEGIN
        Process := 1;
        EventNumber := 6;
        Event6;
      END
    ELSE
      BEGIN
        ConsState := 1;
        Time := TimeNow;
        EventNumber := 7;
        FileWait;
      END;
  END;

PROCEDURE ConsSimulation;
  BEGIN
    CASE EventNumber OF
      7 : Event7;
      8 : Event8;
      9 : Event9;
     10 : Event10;
     END;
  END;

PROCEDURE StatsCollect;
  BEGIN
    FOR I := 1 TO 4 DO AveWait [I] :=
      TimeWait [I] / NumberWait [I] + AveWait [I];
    AveMOE := NumberUnits / TimeNow + AveMOE;
  END;

PROCEDURE Events;
  BEGIN
    CASE Process OF
      1 : ProdSimulation;
      2 : ConsSimulation;
      END;
  END;

PROCEDURE CheckWaitQ;
  BEGIN
    LastWaitPointer := HeadWaitQ;
    ThisWaitPointer := HeadWaitQ;
    WHILE ThisWaitPointer <> NIL DO
      BEGIN
        Process         := ThisWaitPointer^.Process;
        EventNumber     := ThisWaitPointer^.Event;
        Events;

        IF RemoveFlag THEN
          IF ThisWaitPointer = HeadWaitQ THEN
            BEGIN
              HeadWaitQ := HeadWaitQ^.NextPointer;
              LastWaitPointer := HeadWaitQ;
              PushSpareQ (ThisWaitPointer);
              ThisWaitPointer := HeadWaitQ;
              RemoveFlag := FALSE;
            END
          ELSE
            BEGIN
              TempPointer := ThisWaitPointer;
              ThisWaitPointer :=
                ThisWaitPointer^.NextPointer;
              LastWaitPointer^.NextPointer :=
                ThisWaitPointer;
              PushSpareQ (TempPointer);
              RemoveFlag := FALSE;
            END
        ELSE
          BEGIN
            ThisWaitPointer :=
              ThisWaitPointer^.NextPointer;
            IF LastWaitPointer <> HeadWaitQ THEN
              LastWaitPointer :=
                LastWaitPointer^.NextPointer;
          END;
      END;
  END;

PROCEDURE StateTrace;
  BEGIN
    TimeTranslate;

    WRITE (Hours:2, ':', Minutes:2, ':',
      Seconds:6:2, '   ', ProdName [ProdState + 1],
      ' ', ConsName [ConsState + 1],
      ' NumOnBelt = ', NumOnBelt:2,
      ' NumberEmpty = ', NumberEmpty:2);
```

```
    IF BeltFree THEN
      WRITELN ('  Belt Free')
    ELSE
      WRITELN ('  Belt In Use');
  END;

PROCEDURE Initialize;
  BEGIN
    TimeNow := TimeInitial;
    NumOnBelt := 0;
    NumberEmpty := BufferSize;
    BeltFree := TRUE;
    RemoveFlag := FALSE;
    NumberUnits := 0;

    FOR I := 1 TO 4 DO
      BEGIN
        TimeWait [I] := 0;
        NumberWait [I] := 0;
      END;

    ProdState := 1;
    Process := 1;
    EventNumber := 2;
    DupProcess := 1;
    Time := TimeNow + Gamma (1);
    FileEvent;

    ConsState := 1;
    Time := TimeNow;
    Process := 2;
    EventNumber := 7;
    FileWait;

    IF (EtraceFlag) OR (StraceFlag) THEN
      BEGIN
        WRITELN ('BEGIN OF DAY ', NumberMissions:4);
        EventNumber := 1;
        EventTrace;
      END;
  END;

PROCEDURE VarUpdate;

  PROCEDURE RndVarUpdate;
    BEGIN
      REPEAT
        WRITELN (' I    RndVar[1..5, I]');

        FOR J := 1 TO 4 DO WRITELN (J:2,
          RndVar [1, J]:11, RndVar [2, J]:11,
          RndVar [3, J]:11, RndVar [4, J]:11,
          RndVar [5, J]:11);

        WRITELN ('ENTER 1-4 TO INDICATE THE RANDOM',
          ' VARIABLE VALUES ');
        WRITELN ('TO BE CHANGED, OR <space> TO QUIT');
        READ (Key);
        IF Key <> ' ' THEN
          BEGIN
            FOR J := 1 TO 5 DO
              BEGIN
                WRITELN ('ENTER THE NEW VALUE OF ',
                  'RndVar [ ', J,', ', Key, ' ] AS A',
                  ' REAL NUMBER - AND PRESS RETURN');
                READLN (RndVar [J, (ORD(Key) - 48)]);
              END;
          END;
      UNTIL Key = ' ';
      KEY := '*';
    END;

  PROCEDURE ProbUpdate;
    BEGIN
      WRITELN ('ProbIdle = ', ProbIdle:7:5);
      WRITELN ('ENTER NEW IDLE PROBABILITY VALUE AND',
        ' PRESS RETURN');
      READLN (ProbIdle);
      KEY := '*';
    END;

  BEGIN
    REPEAT
      WRITELN (' <1> - RANDOM VARIABLES');
      WRITELN (' <2> - IDLE PROBABILITY');
      WRITELN (' SELECT THE VARIABLES TO UPDATE ');
      WRITELN ('   OR <space> TO EXIT');

      READ (Key);
      CASE Key OF
        '1' : RndVarUpdate;
        '2' : ProbUpdate;
        END;
    UNTIL Key = ' ';
  END;

PROCEDURE ReadInputs;
  BEGIN
    AveMOE := 0;
    FOR I := 1 TO 4 DO AveWait [I] := 0;
    NumberMissions := 1;
    EtraceFlag := FALSE;
    StraceFlag := FALSE;

    REPEAT
      WRITE ('DO YOU WISH TO MODIFY ANY VARIABLES ?',
        ' - Y(es / N(o ');
      READ (Key);
      IF Key IN ['Y', 'y'] THEN VarUpdate;
    UNTIL Key IN ['Y', 'y', 'N', 'n'];

    WRITELN ('ENTER 0, 1, OR 2 TO SELECT EVENT-STATE',
      ' TRACE');
    WRITELN ('  <0> = NO TRACE');
    WRITELN ('  <1> = COMPLETE EVENT-STATE TRACE');
    WRITELN ('  <2> = STATE TRACE ONLY');
    WRITELN ('  - OR <9> TO TERMINATE SIMULATION',
      ' PROGRAM');

    READ (Key);
    IF Key <> '9' THEN
      BEGIN
        IF Key IN ['1', '2'] THEN StraceFlag := TRUE;
        IF Key IN   ['1']    THEN EtraceFlag := TRUE;
        WRITELN ('ENTER NUMBER OF DAYS OF OPERATION',
          ' TO BE SIMULATED');
        WRITELN ('  - AND PRESS RETURN');
        READLN (TotalMissions);
      END
    ELSE
      Stop := TRUE;
  END;

PROCEDURE Report;
  BEGIN
    AveThroughput := AveMOE * 60 / TotalMissions;
    FOR I := 1 TO 4 DO
      AveWait [I] := AveWait [I] / TotalMissions;

    WRITELN ('REPORT OF SIMULATION RUN');
    WRITELN ('THE AVERAGE THROUGH PUT OF',
      ' THE FACTORY IS', AveThroughput:7:2,
      ' UNITS PRODUCED PER MINUTE');
    WRITELN ('BASED ON ', TotalMissions:3,
      ' DAYS OF OPERATION');
```

```
    FOR I := 1 TO 4 DO WRITELN ('THE AVERAGE ',
      'WAITING TIME FOR THE ', WaitName [I],
      ' STATE IS ', AveWait [I]:7:2);
  END;

BEGIN
  RndVarInit;
  ProbInit;
  NameInit;
  QInit;
  Stop := FALSE;
  ReadInputs;
  WHILE NOT Stop DO
    BEGIN
      FOR NumberMissions := 1 TO TotalMissions DO
        BEGIN
          Initialize;

          REPEAT
            IF StraceFlag THEN StateTrace;
            IF HeadEventQ <> NIL THEN PopEventQ;
            Events;
            CheckWaitQ;
          UNTIL (HeadEventQ = NIL) AND
            (HeadWaitQ = NIL);

          StatsCollect;
          UnfileAll;
        END;

      Report;
      ReadInputs;
    END;
END.
```

REFERENCES

1. Clymer, John R., OPERATIONAL EVALUATION MODELING, Unpublished Draft, Department of Electrical Engineering, California State University, Fullerton, California, May 1982.

2. Veit, "The Electronic World - A User's Guide to Computer Languages", Popular Electronics, December 1981.

3. Clark and Koehler, THE UCSD PASCAL HANDBOOK, A REFERENCE AND GUIDEBOOK FOR PROGRAMMERS, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

4. Cooper and Clancy, OH! PASCAL!, AN INTRODUCTION TO PROGRAMMING, W. W. Norton & Company, New York, 1982.

5. "Apple" is a registered trademark of Apple Computer, Inc., Cupertino, California.

6. Fox and Waite, PASCAL PRIMER, Howard W. Sams & Co., Indianapolis, Indiana, 1981.

7. Brainerd, Goldberg, and Gross, PASCAL PROGRAMMING, A SPIRAL APPROACH, Boyd & Fraser Publishing Company, San Francisco, 1982.

8. Quandt, John A., IMPLEMENTATION OF OPERATIONAL MODELING IN PASCAL, 13-22401, Univeristy Microfilms International, Ann Arbor, Michigan, 1984.