Discrete Event Simulation With Pascal

by
Robert M. O'Keefe
Ruth M. Davis

June 1986

TR-86-12

I.6.2 [Simulation and Modeling]: Simulation Languages
I.3.4 [Computer Graphics]: Graphics Utilities

DISCRETE EVENT SIMULATION WITH PASCAL

Robert M. O'Keefe
Department of Computer Science
Virginia Polytechnic Institute and State University
and
Ruth M. Davies
Department of Mathematics, Statistics and Computing
Polytechnic of the South Bank, England

May 7, 1986

Pascal_SIM is a collection of Pascal constants, types, variables, functions and procedures for developing event, activity, three-phase or process orientated discrete-event simulation models. Facilities are provided for queue processing, time advance and event list maintainence, control of entities and resources, random number generation and streams, sampling from parametric and empirical distributions, statistics collection, and visual displays.

Pascal_SIM has been designed as a minimal simulation tool. It includes less than 50 functions and procedures, and totals less than 800 lines of code. It is a basis for programming simulations in Pascal, where users can alter or extend the facilities provided, rather than a simulation programming language.

The majority of Pascal_SIM conforms to the ISO Pascal standard, enabling high portability to be achieved. It can be used immediately with any Pascal that uses the string type descended from UCSD Pascal, for instance Pro Pascal, Turbo Pascal or Sheffield PRIME Pascal. Alteration of a few lines allows for use with any Pascal that provides a different string type, for instance VAX/VMS Pascal.

Key words: Discrete event simulation, visual, Pascal.

Address for correspondance:

Department of Computer Science, Virginia Polytechnic Institute and State University,
Blacksburg, Virginia 24061 U.S.A.

Tel. (703) 961-6075

INTRODUCTION

Discrete event simulation "concerns the modeling of a system as it evolves over time by a representation in which the state variables change only at a countable number of points in time" (1). Normally, the representation is a computer program, and increasingly a program in a Simulation Programming Languages (SPL) such as GPSS (2) or SIMCRIPT (3). However, recent years have seen some resurgence in interest in the use of general purpose programming languages as vehicles for simulation programs. In part this has been due to the increasing availability of good implementations for strongly-typed block structured languages such as Pascal, Ada, and Modula-2. Further, the use of microcomputers has accelerated such interest, since many SPLs and packages are too large for use on microcomputers, or when available on microcomputers, are highly ineffiecient.

A number of Pascal based simulation systems have been produced. The majority of these mimic exisiting SPLs, and do not necessarily make good use of the Pascal that underlies the system. Examples include PASSIM (4), which is based on GPSS, and SIMPAS (5), a SIMCRIPT-like language which is pre-processed into Pascal.

Pascal_SIM is an attempt to provide the basics of a Simulation Programming Language, taking advantage of the main features of Pascal. Those using Pascal_SIM can then add the facilities they need and change the underlying structure if they wish. A further aim is to provide a means whereby students could learn to write simulations in a familiar language using facilities that are well documented and easy to understand.

WORLD VIEWS

At the heart of any SPL or simulation package is a *world view.* This is the methodology for viewing a simulation that is enforced upon the programmer. Pascal_SIM provides two worlds views - the three-phase approach and process description. In the three-phase approach, the programmer programs scheduled events, which can be explicitly scheduled to occur at some time in the future, and conditional events, which occur when certain conditions become true. A simulation is driven by an executive which advances the clock to the next scheduled event, executes it, and then scans all conditional events to ascertain which can be executed.

In process description, the list of activties that each object in the simulation can go through is mapped out, and the executive handles the investigation of the conditions whereby an activity can start, and the scheduling of the end of each activity. Whilst conceptually easier to follow than the three-phase approach, complete implementation of process description, where processes are allowed to interact (called process interaction), requires a co-routines facility. Co-routines are not supported in Pascal, although there have been attempts to extend Pascal with co-routines (6). However, simple process description, where processes can be described, but can not explicitly interact, can be acheived in standard Pascal as is explained later.

VISUAL SIMULATION

Increasingly, the ability to iconically display the ongoing simulation is important (7). Called visual simulation or animation, this aids the programmer in verifying the program, and the model user validating the model. Typically, the amount of effort required to program the

visual display can be as great as that to program the simulation. Thus programming visual displays is supported in Pascal_SIM, and nearly one third of Pascal_SIM is devoted to facilities for producing simple visual displays.

THE STRUCTURE OF A Pascal_SIM PROGRAM

The recommended structure of a three-phase orientated Pascal_SIM program is shown in Figure 1. At the heart of the simulation is the executive, the procedure RUN, which contains the time flow mechanism. Although the structure of this is provided, the user must enter the names of all events into a case statement in this procedure. The number of conditional events MAX_C and the time duration (DURATION) are both arguments. The user must code the events and the procedures INITIALIZE and REPORT; for a visual display DISPLAY and PICTURE must also be coded. INITIALIZE and PICTURE are called once before RUN; they initialize the simulation and the static picture respectively. REPORT should be called after RUN, and should contain any end of run reporting, for example, final statistics prints. DISPLAY is called after every advance of the clock; it should be used to update the visual display as necessary.

The basic object in any simulation is the entity, where entities move through the system engaging in a number of activities. Examples might include patients in a health care simulation, or programs in a computer system simulation. The basis of Pascal_SIM is an entity type, which is a Pascal record thus :-

```
 entity=^an_entity;
an_entity=packed record
        avail:boolean;
        class:class_num;
        col:colour;
        attr,next_B:cardinal;
        time:real;
        end;
```

where the fields of the record represent :-

AVAIL: The availablity of the entity
CLASS: The number of the entities class
COL: The colour of the entity
ATTR: The entities attribute number
NEXT_B: The next bound event or block that the entity will enter
TIME: The time that which this will occur

An entity is always either available, entered in the calendar of future events, or is being used by another entity. If entered in the calendar, AVAIL is false, and NEXT_B and TIME will be set to appropriate values. Thus there are no explicit event notices in Pascal_SIM because an entity contains all relevant event information. Entities are generated with the function NEW_ENTITY, and can be disposed of with the procedure DIS_ENTITY.

Access to entities is acheived through the global variable CURRENT, which always points to the entity that has caused the present event, or else by searching queues of entities.

The attribute number ATTR uniquely indentifies each entity. If further attributes are required they can either be added to entities by using the attribute number to access another data structure, or else by adding in new fields to the entity record and recompiling Pascal_SIM. In complex models, the developer would establish classes, where a class is a list of entities, and both the class and each entity may have attributes. For visual displays, the developer must enter classes into a class table, which holds information on the letter and colour used to represent an entity in the display.

Most simulations involve passive entities, which only serve other entities. These are commonly called resources, and are said to be acquired and released by entities. Examples might include machines in a factory simulation, or a disc drive in a computer system simulation. A resource type, with associated routines, is provided, where resources are collected into a BIN.

The provided functions and procedures of Pascal_SIM are grouped into 11 groups, respectively :-

queue processing
entities and classes
timing and the executive
facilities for process description
resources
error messages
random number generation and streams
sampling distributions
histograms
screen control
visual displays

The interface of Pascal_SIM, ie. all constants, types, global variables and function and procedure heads, is shown in the appendix. A certain excess redundancy is present in the four routines (GIVE_TOP, GIVE_TAIL, TAKE_TOP, TAKE_TAIL) which allow for giving and removing entities to the top and tail of a queue. Use of these allow students to develop First In First Out queueing models without having to explicitly dereference pointers.

AN EXAMPLE - ADMISSION TO HOSPITAL

The example to demonstrate how to program using Pascal_SIM is a hospital simulation, shown as an activity diagram in Figure 2. Two types of patient are admitted to hospital. Those not admitted for an operation undergo a short stay, and then return home. Patients admitted for operation undergo a pre-operative stay, an operation (which requires an open and available operating theatre), followed by a post-operative stay and discharge. Such a simulation is somewhat simplistic, but might be used to investigate various policies regarding bed and operating theatre provision.

```
{ Bound events }

procedure B1;
procedure B2;
  :
  :

{ Conditional events }

procedure C1;
procedure C2;
  :
  :

procedure display;
procedure run(duration:real;max_C:cardinal);
procedure initialize;
procedure picture;
procedure report;

 begin
   :
 initialize;
 picture;
   :
 run( ... , ... );
   :
 report;
   :
 end.
```

Figure 1: The structure of a three-phase Pascal_SIM program.

Listing 1 shows a Pascal_SIM three-phase orientated program for a visual simulation. An example visual display is shown in Figure 3. In the INITIALIZE procedure, the simulation and random number streams are initalized (via MAKE_SIM and MAKE_STREAMS), a BIN called bed with 4 resources is created using MAKE_BIN, and the queues q1,q2,q3 and q4 are initialized using MAKE_QUEUE. The operating theatre is created, and scheduled to close in 8 hours. Note that CAUSE is the scheduling procedure; the first parameter indicates the bound event that will be entered. This has to be entered as an integer, since Pascal does not allow procedure names to be passed as parameters and stored for future calling. Case statements in the executive relate these numbers to procedures calls.

PROGRAMMING THE SIMULATION

Producing a three-phase orientated program for the example commences with the identification of all scheduled and conditional events. For the hospital example these are :-

scheduled

hospital stay only patient arrives;
operation patient arrives;
end hospital stay;
end pre-operative stay;
end operation;
end post-operativestay;
open theatre;
close theatre;

conditional

start hospital stay;
start pre-operative stay;
start operation;
start post-operative stay;

As can be seen in Listing 1, the arrival of a patient of each type is modelled in the procedures PATIENT1_ARRIVES and PATIENT2_ARRIVES. An arrival results in another arrival being scheduled with the CAUSE statement. This is the common way to feed arrivals where the inter-arrival distribution is known. The four activites stay, pre-operative stay, post-operative stay and operation, clearly identifyable in Figure 2, are all modelled by a conditional event that starts the activity, and a scheduled event that ends it. Strictly, the starting of a post operative stay can be a scheduled event, since there are no constraints on its start. However, it is a good idea to model all activities as a pair of conditional and scheduled events, since this aids modularity, and makes the program easier to alter when and if the simulation is extended by introducing further constraints on the start of an activity. Either the closure or opening of the operating theatre is always scheduled. The boolean THEATRE_OPEN is used to record whether or not the theatre is open, and similarly THEATRE_AVAILABLE is set to false if an operation is in progress.

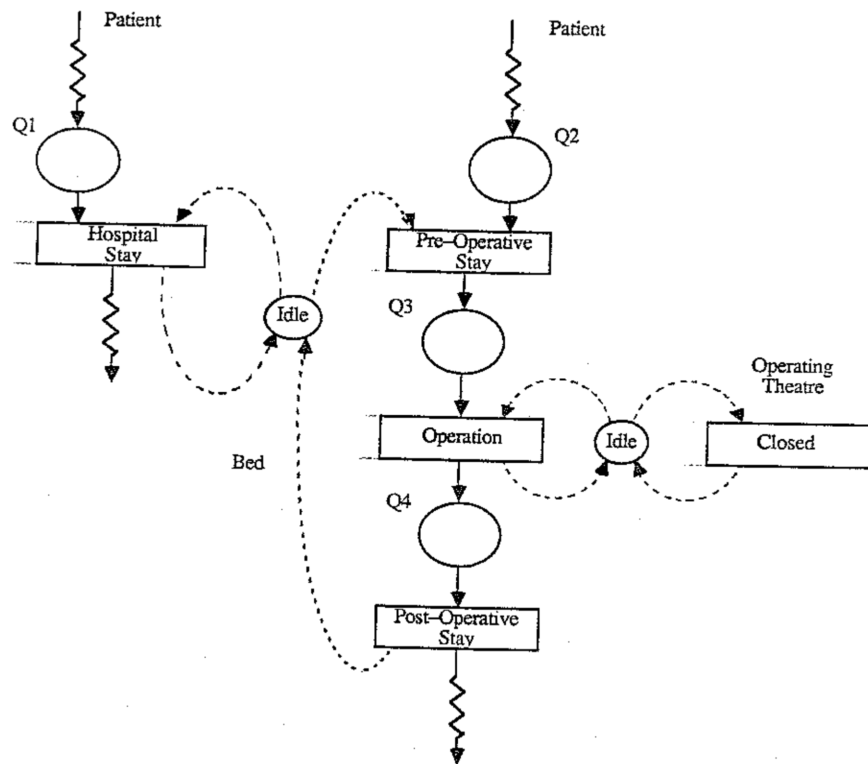

Figure 2: An activity diagram for the hospital simulation.

```
program example;

   var  bed:bin;
        q1,q2,q3,q4:queue;
        theatre:entity;
        theatre_open,theatre_available:boolean;
        { true if theatre is open, available }
        old_tim:real;
        operations_queue:record
                             measure:histogram;
                             last_update:real;
                             end;

procedure patient1_arrives; { stay }
  begin
  give_tail(q1,current);
  move_h(12,2,10,current,white);
  write_queue(22,12,white,q1,20);
  cause(1,new_entity(1,1),uniform(60,140,1));
  end;

procedure patient2_arrives; { operation }
  begin
  give_tail(q2,current);
  move_h(14,2,10,current,white);
  write_queue(22,14,white,q2,20);
  cause(2,new_entity(2,1),uniform(24,48,2));
  end;

procedure end_hospital_stay;
  begin
  return(bed,1);
  move_h(12,40,70,current,white);
  dis_entity(current);
  end;

procedure end_pre_operative_stay;
  begin
  current^.col:=yellow;
  give_tail(q3,current);
  move_v(30,14,20,current,white);
  move_h(20,30,50,current,white);
  write_queue(60,20,white,q3,30);
  end;
```

```
procedure end_operation;
  begin
  theatre_available:=true;
  gotoxy(63,21);write('      ');
  move_v(30,4,10,current,white);
  give_tail(q4,current);
  end;

procedure end_post_operative_stay;
  begin
  return(bed,1);
  move_h(12,40,70,current,white);
  dis_entity(current);
  end;

procedure open_theatre;
  begin
  theatre_open:=true;
  gotoxy(63,20);write('OPEN  ');
  cause(8,current,8);
  end;

procedure close_theatre;
  begin
  theatre_open:=false;
  gotoxy(63,20);write('CLOSED');
  cause(7,current,40);
  end;

procedure start_hospital_stay;
  begin
  while (bed.num_avail>0)
    and (count(q1,true)>0) do
    begin
    acquire(bed,1);
    cause(3,take_top(q1),uniform(20,40,3));
    write_queue(22,12,white,q1,20);
    end;
  end;

procedure start_pre_operative_stay;
  begin
  while (bed.num_avail>0)
```

```
      and (count(q2,true)>0) do
      begin
      acquire(bed,1);
      cause(4,take_top(q2),uniform(5,15,4));
      write_queue(22,14,white,q2,20);
      end;
    end;

  procedure start_operation;
    begin
    while theatre_open and theatre_available
      and (count(q3,true)>0) do
      begin
      theatre_available:=false;
      cause(5,take_top(q3),1);
      gotoxy(63,21);write('IN USE');
      write_queue(60,20,white,q3,30);
      end;
    with operations_queue do
      begin
      log_histogram(measure,count(q3,true),tim-last_update);
      last_update:=tim;
      end;
    end;

  procedure start_post_operative_stay;
    begin
    while (count(q4,true)>0) do
      begin
      cause(6,take_top(q4),uniform(5,10,6));
      end;
    end;

  procedure display;
    var i:cardinal;
    begin
    gotoxy(30,12);write(bed.number-bed.num_avail:1);
    delay;delay;
    for i:=1 to trunc((tim-old_tim)/2) do delay;
    old_tim:=tim;
    gotoxy(1,1);writeln(tim:7:2);
   end { display };

  procedure run(duration:real;max_C:cardinal);
```

```
var c:cardinal;
begin
running:=true;
repeat
  if calendar=calendar^.next then running:=false
  else begin
        display;
        tim:=calendar^.next^.item^.time;
        if duration<tim then running:=false
        else begin
             while (calendar<>calendar^.next) and
                   (tim=calendar^.next^.item^.time)  do
               begin
               calendar_top;
               case current^.next_B of
                 0:;
                 1:patient1_arrives;
                 2:patient2_arrives;
                 3:end_hospital_stay;
                 4:end_pre_operative_stay;
                 5:end_operation;
                 6:end_post_operative_stay;
                 7:open_theatre;
                 8:close_theatre;
               end;
               end;
             for c:=1 to max_C do
               case c of
                 1:start_hospital_stay;
                 2:start_pre_operative_stay;
                 3:start_operation;
                 4:start_post_operative_stay;
               end;
             end;
      end
until not running;
end { run };

procedure initialize;
begin
make_sim;
make_streams;
make_bin(bed,4);
make_queue(q1);make_queue(q2);
```

```
    make_queue(q3);make_queue(q4);
    with operations_queue do
      begin
      make_histogram(measure,1,1);
      last_update:=0;
      end;
    { create theatre }
    theatre:=new_entity(3,1);
    theatre_open:=true;
    theatre_available:=true;
    cause(8,theatre,8);
    end { initialize };

procedure picture;
  var i:cardinal;
  begin
  enter_class(1,'s',blue);
  enter_class(2,'o',blue);
  for i:=1 to 80 do writeln;
  write_block(28,10,32,14,magenta);
  write_block(60,18,70,23,magenta);
  set_foreground(yellow);
  gotoxy(4,11);write('Hospital stay only');
  gotoxy(4,15);write('Operation');
  gotoxy(32,8);write('Beds in use');
  gotoxy(60,15);write('Operating');
  gotoxy(60,16);write('Theatre');
  reset_colours;
  end { picture };

procedure report;
  begin
  print_histogram(output,operations_queue.measure,false,60);
  end { report };

begin
initialize;
picture;
cause(1,new_entity(1,1),0);
cause(2,new_entity(2,1),0);
old_tim:=0;
run(24*30*12,4);
report;
reset_colours;
```

```
end.
```

Listing 1: A three-phase Pascal_SIM simulation for the hospital example.

PROGRAMMING THE VISUAL DISPLAY

The visual display is composed of two parts - a static background picture which is written to the console once prior to the simulation run, and a dynamic display which moves over the static picture. The dynamic display can be updated either within an event, bound or conditional, or following a time beat (where one or more events will have been executed) in the procedure DISPLAY.

Even with using text to program simple iconic visual displays, a minimum amount of screen control is essential. Cursor addressing must be possible; for colour displays the ability to set both foreground and background colour is necessary. Many terminals provide both of these, and thus the visual display routines are highly portable.

A static picture is created in the procedure PICTURE. Entity classes 1 and 2 (respectively hospital stay only and operation patients) are entered in the CLASS_TABLE with letters 's' and 'o'. Both will appear blue, unless the field COL in the entity record has been set to a colour - this overrides the CLASS_TABLE entry. To provide a background, blocks coloured magenta are entered in the display, and some simple annotation is provided using the GOTOXY procedure in Pascal_SIM and the standard Pascal procedure WRITE.

The procedure DISPLAY provides for updating of the dynamic display after a time beat. The number of BEDs in use (BED.NUMBER-BED.NUM_AVAIL) is written. At this point, the display is completely up to date. The simulation is then delayed relative to the time before the new time beat (TIM-OLD_TIM). If this is not done, the display advances too quickly for comfortable viewing. The new clock time TIM is then written to the display, and the simulation (and thus the part of the picture generated within events) can continue.

Most of the visual display statements are embedded in the events. For instance, when a hospital stay only patient arrives, the following occurs (see procedure PATIENT1_ARRIVES) :-

put patient on a queue for a bed
show the arrival of the patient by horizontal movement
display the hospital stay only queue for beds
cause the arrival of a new hospital stay only patient

The procedures MOVE_H (move horizontal) and MOVE_V (move vertical) are the main methods for moving entities across the display. The letter representing the entity is moved on the specified background colour either horizontally or vertically. Note that when ready

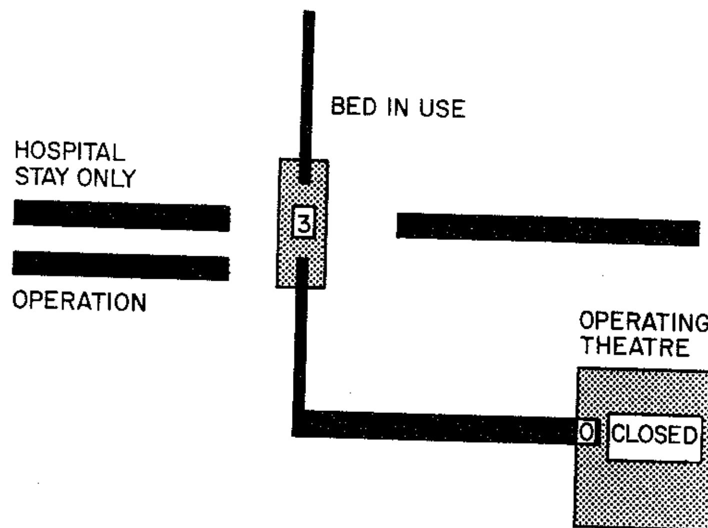

Figure 3: An example visual display. A patient, represented by the letter 'o', is waiting for the operating theatre to open. Three beds are in use; the present clock time is 783.06 time units.

```
mean =     0.45  variance =      0.30  sd =     0.55
min =     0.00  max =     2.00

under    ************************************************   5030.20
   1.00  *******************************                    3366.76
   2.00  *                                                   243.04
   3.00
   4.00
   5.00
   6.00
   7.00
   8.00
   9.00
  10.00
  11.00
  12.00
  13.00
  14.00
  15.00
over
```

Figure 4: A histogram for the size of the operating theatre queue.

for an operation, in procedure END_PRE_OPERATIVE_STAY, operation patients have their colour set to yellow, and will thus appear yellow on the screen from that point on.

STATISTICS COLLECTION

Collection of statistics is acheived through a HISTOGRAM type. The programmer must explicitly create histograms with MAKE_HISTOGRAM, arrange to have data logged to them with LOG_HISTOGRAM, and print them to a file with PRINT_HISTOGRAM. A histogram can be of one of two types. State histograms record absoloute values, whereas time weighted histograms result in the calculation of parameters being wieghted by the times that an observation persisted for.

Listing 2 includes the use of a time weighted histogram to record the queue length at the operating theatre. A record called OPERATIONS_QUEUE is declared with two fields MEASURE (the histogram) and LAST_UPDATE (the last time at which an observation was recorded). The queue length is logged in the procedure START_OPERATION, after any change in queue length due to the start of an operation. An example final histogram is shown in Figure 4. As a time weighted histogram, accumulated cell totals represent time duration. Thus, for instance, the queue was empty for 5030.20 time units.

A PROCESS DESCRIPTION VERSION

The hospital example is recoded as three processes for hospital stay only patients, operation patients, and the operating theatre, in Listing 2. Note that DISPLAY, INITIALIZE and PICTURE would be identical to those in Listing 2 and have thus been omitted, as has the statistics collection.

Process orientated simulations in Pascal_SIM are written by splitting up a procedure into a number of different blocks using a case statement. Entities advance from block to block; a procedure can be 'reactivated' by being called and a block (ie. a branch of the case statement) being attempted. A procedure called BRANCH is used to immediately cause an entity to attempt a new block. Note that the first parameter of a CAUSE statement now specifies a block number rather than an event, and an entities class number identifies its process. The need for the programmer to split a procedure up with a case statement, to explicitly specify when a process must be suspended (this is done here using a repeat until loop with a flag called finished), and to explicitly test if entry to a block is possible, makes the programming fairly ungainly and error prone compared to true process interaction languages. It does, however, allow for process description modelling in Pascal.

In the process description executive, which is modelled on the GPSS executive, an entity is either entered in the calendar or on a chain of suspended entities. Queue priority is implicit, dependent upon an entity's position in the suspended chain. This means that the developer of a visual simulation must introduce dummy queues at various points in a process so that the queues can be written to the picture. This has not been done in Listing 2, therefore the queues displayed using WRITE_QUEUE in Listing 1 would not be present in the picture

```
program example;

   var  bed:bin;
        q1,q2,q3,q4:queue;
        theatre:entity;
        theatre_open,theatre_available:boolean;
        { true if theatre is open, available }
        old_tim:real;

procedure patient1;
  var finished:boolean;
  begin
  finished:=false;
  repeat
    case current^.next_B of
      1:begin
        move_h(12,2,10,current,white);
        cause(1,new_entity(1,1),uniform(30,60,1));
        branch(2);
        end;
      2:begin
        if (bed.num_avail>0) then
```

```
            begin
            cause(3,current,uniform(20,40,3));
            acquire(bed,1);
            end;
          finished:=true;
          end;
        3:begin
          return(bed,1);
          move_h(12,40,70,current,white);
          finished:=true;
          remove_entity;
          end;
      end;
      until finished;
      end { process for patient 1 };

procedure patient2;
  var finished:boolean;
  begin
  finished:=false;
  repeat
    case current^.next_B of
      1:begin
        move_h(14,2,10,current,white);
        cause(1,new_entity(2,1),uniform(12,20,2));
        branch(2);
        end;
      2:begin
        if (bed.num_avail>0) then
          begin
          cause(3,current,uniform(5,15,4));
          acquire(bed,1);
          end;
        finished:=true;
        end;
      3:begin
        current^.col:=yellow;
        move_v(30,14,20,current,white);
        move_h(20,30,50,current,white);
        branch(4);
        end;
      4:begin
        if theatre_open and theatre_available then
          begin
```

```
            theatre_available:=false;
            cause(5,current,1);
            gotoxy(63,21);write('IN USE');
            end;
          finished:=true;
          end;
        5:begin
          theatre_available:=true;
          gotoxy(63,21);write('      ');
          move_v(30,4,10,current,white);
          cause(6,current,uniform(5,10,6));
          finished:=true;
          end;
        6:begin
          return(bed,1);
          move_h(12,40,70,current,white);
          finished:=true;
          remove_entity;
          end;
    end;
    until finished;
    end { process for patient 2 };

procedure schedule_theatre;
  begin
  case current^.next_B of
    1:begin
      theatre_open:=true;
      gotoxy(63,20);write('OPEN  ');
      cause(2,current,8);
      end;
    2:begin
      theatre_open:=false;
      gotoxy(63,20);write('CLOSED');
      cause(1,current,40);
      end;
    end;
  end { schedule theatre };

procedure display;
  begin
   :
  end { display };
```

```
procedure run(duration:real);
  var c:link;
      remove,changed:boolean;
      e:entity;
      i:cardinal;

  procedure reactivate(scanning:boolean);
    var present:cardinal;
    begin
    remove:=false;on_calendar:=false;
    with current^ do
      begin
      present:=next_B;
      case class of
        1:patient1;
        2:patient2;
        3:schedule_theatre;
      end;
      if scanning then
        begin
        if on_calendar then
          begin
          remove:=true;changed:=true;
          end
        else
        if present<>next_B then
          begin
          remove:=true;changed:=true;
          give_tail(suspended_chain,current);
          end;
        end
       else begin
            if next_B<>0 then
              if not on_calendar then
                give_tail(suspended_chain,current);
            end;
      if next_B=0 then dis_entity(current);
      end;
   end { reactivate };

begin
running:=true;
repeat
  if calendar=calendar^.next then running:=false
```

```
     else begin
          display;
          tim:=calendar^.next^.item^.time;
          if duration<tim then running:=false
          else begin
               while (calendar<>calendar^.next) and
                     (tim=calendar^.next^.item^.time)  do
                 begin
                 calendar_top;
                 reactivate(false);
                 end;
               repeat
               changed:=false;
               c:=suspended_chain^.next;
               i:=1;
               while c<>suspended_chain do
                 begin
                 current:=c^.item;
                 reactivate(true);
                 c:=c^.next;
                 if remove then e:=take(suspended_chain,c^.pre);
                 end;
               until not changed;
               end;
          end
   until not running;
   end { run };

procedure initialize;
  begin
   :
  end { initialize };

procedure picture;
  begin
   :
  end { picture };

procedure report;
  begin
   :
  end { report };

begin
```

```
initialize;
picture;
cause(1,new_entity(1,1),0);
cause(1,new_entity(2,1),0);
old_tim:=0;
run(24*30*12);
report;
reset_colours;
end.
```

Listing 2: A process description version of the hospital simulation.

PORTABILITY AND IMPLEMENTATION

Considerable portability is achieved by close adherence to the ISO Pascal standard. Only two non-standard Pascal facilities are used - the use of an underscore in names (which can easily be edited out), and the use of a string type. However, most Pascal implementations provide a string type, and Pascal_SIM can be implemented without change under any Pascal that uses the string type and associated functions descended from UCSD Pascal. Examples include Turbo Pascal, Pro Pascal, and the Pascal compiler for PR1ME systems produced at the University of Sheffield in England. If a string type is defined differently, or different functions are provided, a few alterations are necessary. For instance, in VAX/VMS Pascal, the string type is VARYING ARRAY OF CHAR rather than STRING, and strings are concatenated directly using the addition operator rather than a CONCAT function. If strict ISO Pascal is followed, and a PACKED ARRAY OF CHAR has to be used, then only one procedure is unuseable. This is PRINT_HISTOGRAM, which prints histograms to a text file.

Pascal_SIM is normally implemented by some method of prior compilation. Methods include adding the functions and procedures to a library and the variables to a common area (this is the method of implementation in Pro Pascal), production of a unit or module, containing all of Pascal_SIM, that is then put in a library (for instance, UCSD Pascal), or by a similar method (for instance, implementation in VAX/VMS Pascal is acheived by production of an environment file for the constants, types and variables, and an associated module for the functions and procedures). Thus the facilities are available to any Pascal program by simple reference to the library, unit or whatever. Pascal_SIM has been used extensively with Turbo Pascal, which provides no facilities for separate compilation. Here the programmer must recompile Pascal_SIM with the simulation.

To implement Pascal_SIM, it is necessary to set up the screen control codes within a number of procedures. Many terminals can be made to accept ANSI screen control codes (for instance, IBM-PC monitors) or use an extension of ANSI (for instance, DEC VT100 and

VT240). Thus ANSI screen control (and the extended ANSI descended from Textronix for colour text) is frequently sufficient. Additional copies of some screen control and visual display routines are provided for use with Turbo Pascal, which call the screen control routines built into Turbo Pascal.

CONCLUSIONS

The authors have mainly used Pascal_SIM with Turbo Pascal and VAX/VMS Pascal. Pascal_SIM, Turbo Pascal, a colour monitor, and an IBM- PC/XT or AT allow visual simulations of resonable display quality to be developed and run. For statistical experimentation, the model can then be ported to a VAX, and the Pascal_SIM statements relating to the visual display replaced by statements for statistics collection using histograms. Shortly, Pascal_SIM will be rewritten in Modula-2, where the built in co-routine facility will allow for proper and neater process interaction simulation.

Programming visual simulations can be time consuming, and typically in the hospital example there are more programmed statements relating to the display than to the logic of the simulation. This is true for other programming language orientated visual simulation systems, for example the FORTRAN based system SEE-WHY (8).

Having both the three-phase approach and process description world views in one package is very useful for teaching. Students can program models using a both views, and thus obtain a better understanding of frameworks for simulation model building than when using a single view.

Many simulations are still programmed in FORTRAN (9). Increasingly students of science and engineering subjects are learning Pascal as their main programming language. They will undoubtedly want to write simulations in Pascal. Pascal_SIM provides a structure and the facilities to do this.

Acknowledgements

Many of the ideas in Pascal_SIM can be traced back to a Pascal based system produced by John Crookes at the University of Lancaster, England.

This paper was completed whilst the first author was on leave from the Board of Studies in Management Science, University of Kent at Canterbury, England.

The following are trademarks :

Pro Pascal: Prospero Software Limited
Turbo Pascal: Borland International
VAX/VMS: DEC
IBM-PC: IBM
UCSD Pascal: Regents of the University of California
MS-DOS: Microsoft
Ada: United States Department of Defence

REFERENCES

1. A. Law and D. Kelton, *Simulation Modelling and Analysis*, Mc-Graw-Hill, New York, 1982, pp. 4.

2. T. Schriber, *Simulation Using GPSS*, Wiley, New York, 1974.

3. E.C. Russell, *Building Simulation Models with SIMSCRIPT II.5*, C.A.C.I., Los Angeles, 1983.

4. D. Uyenso and W. Vaessen, "PASSIM: a discrete-event simulation package for Pascal", *Simulation* **35**, 183-190 (1980).

5. R.M. Bryant, "SIMPAS: a simulation language based on Pascal", *Proceedings of the 1980 Winter Simulation Conference (Oren, Shub and Roth, eds.).* The Society for Computer Simulation, La Jolla , 1980.

6. J. Kriz and H. Sandmayer, "Extension of Pascal by co-routines and its application to quasi-parallel programming and simulation", *Software - Practice and Experience* **10**, 773-789 (1980).

7. P.C. Bell, "Visual Interactive Modelling in Operational Research: successes and opportunities", *J. Opl. Res. Soc.* 36, 975-982 (1985).

8. E. Fiddy, J.G. Bright and R.D. Hurion, "SEE-WHY: interactive simulation on the screen", *Proceedings of the Institute of Mechanical Engineers* **c293/81**, 167-172 (1981).

9. D.P. Christy and H.J. Watson (1983), "The application of simulation: a survey of industry practice", *Interfaces* **13**, 47-52 (1983).

APPENDIX

```
const  max_cell_num=16;
       max_stream_num=32;
       max_class_num=256;
       max_sample_num=20;
       max_string_length=80;
       delay_num=2000;

type  a_string=string[max_string_length];
      cardinal=0..maxint;

      colour=(nul,black,red,green,yellow,blue,magenta,cyan,white);

      stream_num=1..max_stream_num;
```

```
    cell_num=0..max_cell_num;
    class_num=1..max_class_num;
    sample_num=1..max_sample_num;
    string_length=1..max_string_length;

    entity=^an_entity;
    link=^a_link;
    a_link=record
          next,pre:link;
          item:entity;
          end;
    queue=link;

    an_entity=packed record
            avail:boolean;
            class:class_num;
            col:colour;
            attr,next_B:cardinal;
            time:real;
            end;

    bin=record
        number,num_avail:cardinal;
        end;

    histogram=record
            cell:array[cell_num] of real;
            count,width,base,total,sosq,min,max:real;
            end;

    lookup_table=array [1..max_sample_num,1..2] of real;

var tim:real;
    current:entity;
    calendar:queue;
    on_calendar:boolean;
    suspended_chain:queue;
    running:boolean;
    original_seeds,seeds:array [stream_num] of cardinal;
    class_table:array [class_num] of
              record
              let:char;col:colour;
              end;
```

```
{ queue processing }
procedure make_queue(var l:queue);
procedure give(l:queue;t:link;i:entity);
function  take(l:queue;t:link):entity;
procedure give_top(l:queue;i:entity);
procedure give_tail(l:queue;i:entity);
function  take_top(l:queue):entity;
function  take_tail(l:queue):entity;

{ entities and classes }
function  new_entity(c:class_num;a:cardinal):entity;
procedure dis_entity(e:entity);
procedure make_class(var c:queue;n,size:cardinal);
function  count(var l:queue;a:boolean):cardinal;

{ timing and the executive }
procedure make_sim;
procedure cause(nb:cardinal;e:entity;t:real);
procedure calendar_top;

{ facilities for process executive }
procedure branch(next:cardinal);
procedure remove_entity;

{ resources }
procedure make_bin(var from:bin;n:cardinal);
procedure acquire(var from:bin;n:cardinal);
procedure return(var from:bin;n:cardinal);

{ error messages }
procedure sim_error(s:a_string);

{ random number generator and streams }
procedure make_streams;
procedure rnd(s:stream_num):real;

{ sampling distributions }
function normal(m,sd:real;s:stream_num):real;
function log_normal(m,sd:real;s:stream_num):real;
function poisson(m:real;s:stream_num):cardinal;
function negexp(m:real;s:stream_num):real;
function uniform(l,h:real;s:stream_num):real;
procedure make_sample(var sample_file:text;var table:lookup_table);
function sample(table:lookup_table;s:stream_num):real;
```

```
{ histograms }
procedure reset_histogram(var h:histogram);
procedure make_histogram(var h:histogram;cell_base,cell_width:real);
procedure print_histogram(var pr:text;h:histogram;state:boolean;len:cardinal);
procedure log_histogram(var h:histogram;where,what:real);

{ screen control }
procedure make_screen;
procedure gotoxy(x,y:cardinal);
procedure clear_screen;
procedure set_foreground(c:colour);
procedure set_background(c:colour);
procedure reset_colours;

{ visual displays }
procedure delay;
procedure make_class_table;
procedure enter_class(n:class_num;l:char;c:colour);
procedure write_entity(x,y:cardinal;e:entity);
procedure write_queue(x,y:cardinal;
                b:colour;q:queue;max_length:cardinal);
procedure write_block(x1,y1,x2,y2:cardinal;b:colour);
procedure move_v(x,y1,y2:cardinal;e:entity;b:colour);
procedure move_h(y,x1,x2:cardinal;e:entity;b:colour);
procedure write_time;

{ user written routines }
procedure display;
procedure initialize;
procedure picture;
procedure report;

{ simulation executive }
procedure run(duration:real;max_C:cardinal);
```