Proceedings of the 1987 Winter Simulation Conference
A. Thesen, H. Grant, W. David Kelton (eds.)

WRITING SIMULATIONS FROM SCRATCH: PASCAL IMPLEMENTATIONS

Arne Thesen
Department of Industrial Engineering
University of Wisconsin - Madison
Madison, WI, 53706, USA

ABSTRACT

Techniques for implementing simulation models in Pascal are discussed. Special emphasis is placed on the development of efficient data structures and random number generators. Source codes for efficient but not commonly available algorithms are provided. A floppy disk containing all procedures discussed in the paper is available from the author.

I. INTRODUCTION

A significant number of new simulation languages and subroutine packages are introduced each year. Common for most of these languages are the facts that a) their *usage* is (usually) well documented; and, b) their *internal design* is not documented or explained at all.This discrepancy may be one of the reasons why there is such a proliferation of "home made" simulation languages. The only way to have a language that one understands well enough to be able to modify it is to write one's own language. Unfortunately, language developers seldom have the time or expertize to search out and implement state of the art solutions from the variety of different disciplines involved in the implementation of a sophisticated simulation language.

A. Special features of simulation programs

Simulation programs have the unique feature that at least 90% of the code in any one application is general purpose code and that at most 10% of the code is specific to any one application. For example, the Procedure shown in Prog 1 is (a simplified version of) the user written model of a simulation designed to determine the expected weekly maximum queue size for a clinic lobby in Madison Wisconsin. The remainder of the simulation program (2300 lines) is general purpose code, used by this and other simulation models. Among the tasks performed by this code are:

1. Time Keeping and Event Scheduling
2. Random Variate Generation
3. Set Management
4. Keyboard Monitoring
5. Data Collection and Reporting
6. Real Time Graphics
7. Error Checking
8. Management of Modelling Constructs.

```
Procedure UserModel;
var
   Client           : EntityPtrType;
   InfoServers      : Integer;
   MeanInfoTime     : Real;
   MeanInfoInterval: Real;
begin

{ define arrival process}
  MeanInfoTime  := 1.00;  {Minutes}
  Recurrent('A',MeanInfoInterval,1);

{ Inquiry booth }
  MeanInfoInterval:= 0.3;
  InfoServers:=trunc(MeanInfoTime
               /MeanInfoInterval) + 1;
  MakeWorkStation(1,1,2,0,
       InfoServers,'Desk');
  SetLabel(1,'WS 1 Info Queue');
  SetLabel(2,'WS 1 Info Clerks');
  ProcessingTime(1,1,MeanInfoTime,2);

{ event scheduling}
  Repeat
   NextEvent;
   case EventCode of
    'A':begin
         MakeEntity(Client,' ',1,nil);
         EnterWorkStation(1,'',Client);
        end;
   end;
  Until done;
end;
```

Prog.1: Pascal based model of a simple queuing system. Note the use of model building blocks such as Workstations and Recurrent Event Streams.

Linstrom and Skansholm(1981) discusses the general problem of designing simulation software. It is not reasonable to expect the end-user to understand how these tasks are being carried out. However, the end-user should expect the system to implement these functions correctly and efficiently (our experience suggest that this may not always be the case).

B. Special difficulties with Pascal.

Modern programming languages such as Ada and Modula 2 include features such as separately complied modules, initializors and static variables that make it relatively easy to implement general purpose simulation programs (Thesen and Sun (1985), L'Eculier (1987)) and systems (Livney(1987)). Pascal on

```
Program  S(input,output);

{----------PUBLIC DEFINITIONS----------}

CONST {count of sets ,work stations etc}
TYPE  (Entity records, event notices}
VAR    Current time, trace flags,
          urrent  Event code, seeds}
Forward {all user callable routines}

{-------USER MODEL-----------------}

Procedure userModel; {user written}

User's definitions

Begin
  User's code.
end;

{-----------PRIVATE DEFINITIONS---------}

{Everything beyond this point is unknown
     to the user}

CONST    {Count of graphic tokens,
          multipliers etc}}
TYPE     {Set headers, Node records, data
             collection records etc
VAR      {all variables not explicitly
           needed  by the user}

{Systems Procedures}

Begin
  StartSimulation
end.
```

Prog 2: Structure of a Pascal based simulation program. The user written model is placed up front. Forward declarations are used to give the user access to systems procedures defined later in the program.

the other hand was not designed with large, multi programmer systems in mind, and standard Pascal introduces many obstacles to good (simulation) program design. Among these are:

1. Only globally declared variables remain defined through the simulation;
2. Separately complied subroutines are not allowed.
3. Pointers to records of different types are of different type;
4. Procedure calls must always have the same number of parameters.

Many extensions to standard Pascal are provided by different compilers. For example, some Pascal compilers allow declarations and definitions to be placed wherever procedure statements can be placed. This feature can be exploited to hide most systems variables from the user. This is illustrated in Prog. 2 where we show the structure of a program where those system variables and procedures that the user should know about are defined *before* the user written model, and everything else is defined *after* this program. All systems procedures follow the user defined model. Global forward declarations are used to tell the user about those language routines that the user may call. Everything else is outside the scope of the user model (and hence protected from his/her intervention).

C. The rest of this paper

It is the purpose to this paper to assist would-be language developers by presenting a survey of current approaches to some of the more important and difficult design problems facing language developers. In doing the research for this paper, we developed a Pascal based simulation language S.PAS. This language, which illustrates all the points discussed in this paper (and many others such as model building blocks (i.e. workstations, recurrent event streams), additional random number generators and separately compiled modules using TurboParcal 4.0) . S.PAS is not intended to compete with many of the excellent Pascal based simulation languages (e.g.Bryant(1980), Uyeno and Vaessen(1980), Seila(1986), Barnett(1986), Mallroy, and Soffa (1986), O'Keefe and Davies (1986)) currently available. Copies of S.PAS are available from the author.

In section two of this paper we discuss the problem of event set management. An empirical evaluation of different approaches is given and the code for an efficient algorithm for tree structured set management is given. In section three we present efficient algorithms for the generation of random variates from the uniform, exponential, normal and gamma distributions. S.PAS also illustrates the use of model building blocks and real-time graphics . These subjects are not covered in this paper due to space limitations.

II. EVENT SET MANAGEMENT

A simulation program may be thought of as a specialized data base management program. *Records* are used to represent entities and events, and *pointers* are used to link together records of similar types such that a *logical ordering* of records is maintained. For example, as shown in Figure 1, records representing event notices are linked together by pointers such that event notices are maintained in increasing order.

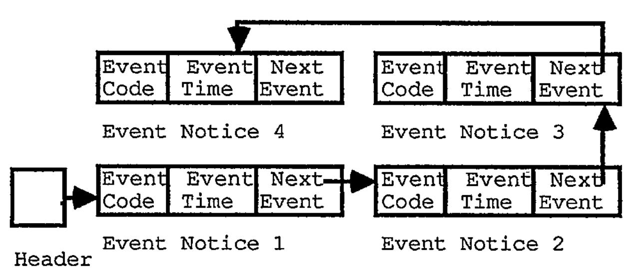


Figure 1: A simple data structure for the event set

A. Need for efficiency

A significant fraction of total computer time and a substantial amount of computer code is usually devoted to the maintenance of the event set. The most time consuming part of this task is to find the proper position in the event set for insertion of new event notices. If new events are equally likely to be inserted in any position in the event set, then, on the average, half of the set must be searched for each insertion if the data structure used in Figure 1 is used. This searching can be extremely time consuming for large event sets. (However, event notices are not likely to be created in this fashion, and a somewhat faster search scheme may be possible if it is known if an event is likely to be inserted towards the beginning or the end of the set).

Given the expensive nature of event set management using the data structure in Figure 1, most commercial languages use a more sophisticated structure. Many such structures have been suggested. McCormac and Sargent (1981) analyzed algorithms available at that time. Four algorithms performed well:

Binary Search indexed-List (Henriksen (1977))
Modified Heap, (two versions).
Indexed-List (Vaucher & Duval (1975))

Several additional approaches have been suggested since that time (most notably Kingston (1984) and Sleator and Tarjan(1985)). The relative merit of these structures is still open for debate. For example there is no universal agreement on the battery of tests to which algorithms should be subjected (see for example Vaucher's (1986) letter to the editor commenting on a recent paper by Jones (1986)). The tradeoffs in selecting a data stru-cture are between simplicity (i.e.single linearly linked list), presence of underlying theory (i.e.splay tree) and ease of implementation. Common for all efficient algorithms is the fact that they reduce the length of the path being searched by introducing a tree-like data structure. This tree can be explicit (Section II.B) or it can be implicit (for example Simscript II.5 uses multiple lists, and a branching mechanism is used to find the proper list).

B. A Tree structure for event sets

A fairly elaborate, but efficient data structure for event set management is given in Figure 2. This structure has the following features:

- Several classes of event codes are recognized
  - -- User codes are conventional event codes
  - -- Systems codes are intercepted by the event manager, and the corresponding events are handled without user knowledge. (Used to generate recurrent event streams, flush work stations, graphics, etc.).
- Three classes of records are used
  - -- The event notice
  - -- The affiliated entity (if any)
  - -- The node

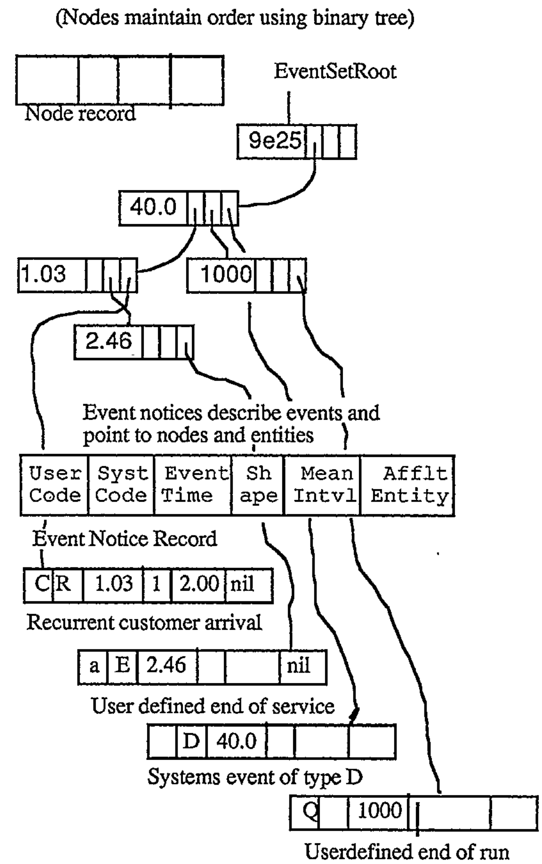


Figure 2: Structure of Event Set Data Base. Note use of separate records to maintain order

We use a separate node record to maintain order in the set. Identical node records are used to maintain order in other sets (such as queues etc) .This enables us to user the same code and data structure to maintain order in all our sets.

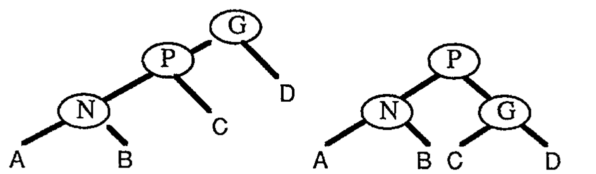
C. Using rotations to balance event trees

A problem with the data structure proposed above is that the resulting binary tree may evolve in an unbalanced fashion. In fact it is theoretically possible for the tree to degenerate to a linearly linked list (although this is extremely unlikely in practice). Several techniques have been proposed for making adjustments in the tree such that it remains balanced at all times. A particularly attractive approach involving automatic rotations following each insertion is suggested by Sleator and Tarjan (1985). Their approach is roughly a three step procedure:

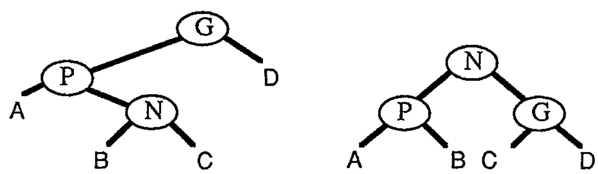
1. Insert the new event notice as a leaf at the proper place in the tree.

2. Determine if rotations are required (they are if the new node is more that four levels from the root).

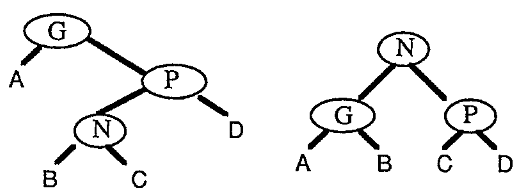
3. Recursively rearrange the structure of the subtree starting at the new node's grand parent according to the rotation rules given in Figure 3.



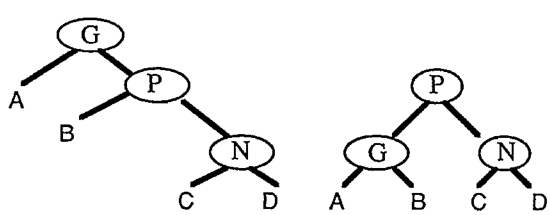
Case 1: New node (N) is left child of parent (P). Parent is left child of grandparent (G).



Case 2: New node (N) is right child of parent (P). Parent is left child of grandparent (G).



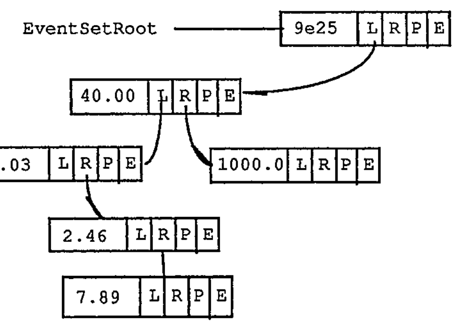
Case 3: New node (N) is left child of parent (P). Parent is right child of grandparent (G).



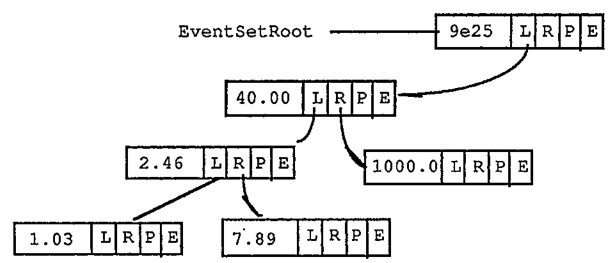
Case 4: New node (N) is right child of parent (P). Parent is right child of grandparent (G).

Figure 3: The effect of splay rotations on the event set. Rotations are intended to reduce the depth of the tree. The applicable rotation depends on the path from the newly inserted node to the root.

An example of a single application of these rotations is given in Figure 4 . A Pascal program performing the initial insertion of an event notice and the subsequent rotations is presented in Prog 3.



Event set after event at time 7.89 is inserted



Event set after single rotation

Figure 3: Effect of rotation on an event set

D. Evaluation

Some authors argue that the complexity and high setup cost of elaborate set management procedures cause them to be impractical for applications with small event set. To develop an understanding for these issues, we measured the time required to insert events into event sets of different sizes. The results are Summarized in Table 1.

Data structure	Size of Event set					
	1	25	50	100	250	500
Simple linked list	8	19	31	57	132	230
Circular List	7	15	26	45	100	189
Unbal. Binary Tree	8	12	14	16	17	18
Splay Rotated Tree	9	15	17	23	26	44

Table 1: Time in Seconds to process 2000 events.for a simple queuing simulation using four different data structures

```
{-------------------------------------}
Procedure LinkNode(var RootPtr:
     nodePtrType; NewPriority : real);
{-------------------------------------}
{ Attache the new node as a leaf in}
{  the binary tree rooted by root }

VAR Temp        : NodePtrType;
    NewNodePtr  : NodePtrType;
    done        : boolean;
    P           : NodePtrType;
    GP          : NodePtrType;
    LocalRoot   : NodePtrType;
    Rotate      : integer;

Procedure RotateRequired;
{ Determine direction of last two links}
begin
  p          :=NewNodePtr^.Parent;
  gp         :=p^.Parent;
  LocalRoot  := gp^.parent;
  Rotate     :=0;
  if localRoot <> RootPtr then
    if p^.left = NewNodePtr then
      if gp^.left = p then
         {p.left =NewNode; gp^.left = p}
        Rotate:= 1      else
         {p.left =NewNode; gp^.right =p}
        Rotate := 2     else
      if gp^.left = p then
          {p.right =NewNode; gp^.left=p}
        Rotate := 3 ;
       else
          {p.right =NewNode;gp^.right=p}
         Rotate := 4; {end;

Procedure DoRotate;
Var a,b,c,d : NodePtrType;
Begin
  if LocalRoot <> RootPtr then begin
    Case Rotate of
     1:Begin
          C          := P^.Right;
          P^. Right := GP;
          P^. Parent:= LocalRoot;
          GP^.Parent:= P;
          GP^.Left   := c;
          if c<>nil then c^.parent:=GP;
          if LocalRoot^.left = GP  then
            LocalRoot^.left := p
          else
            LocalRoot^.Right := p;
        end;
     2 :Begin
          B         := NewNodePtr^.left;
          C         := NewNodePtr^.Right;
          D            := P^.Right;
          NewNodePtr^. left := gp;
          NewNodePtr^. Right:= p;
          NewNodePtr^.Parent:=LocalRoot;
          GP^.Right            := B;
          P^. Left             := C;
          if b <>nil then B^.Parent:=GP;
          if c <>nil then C^.Parent:= P;
          P^. Parent  := NewNodePtr;
          GP^.Parent  := NewNodePtr;
          if d<>nil then d^.Parent :=p;
          if LocalRoot^.left = GP  then
            LocalRoot^.left:=NewNodePtr
          else
           LocalRoot^.Right:=NewNodePtr;
        end;
     3:Begin
          B          :=NewNodePtr^.left;
          C          :=NewNodePtr^.Right;
          NewNodePtr^.left  :=p;
          NewNodePtr^.Right := gp;
          NewNodePtr^.Parent:= LocalRoot;
          P^.Right          :=b;
          P^.Parent         :=NewNodePtr;
          GP^.Left   :=c;
          GP^.Parent := NewNodePtr;
          if b <> nil then B^.Parent:=P;
          if c <> nil then C^.Parent:=gP;
          if LocalRoot^.left = GP  then
            LocalRoot^.left := NewNodePtr
          else
            LocalRoot^.Right:=NewNodePtr;
        end;

     4:Begin
          B          := P^.Left;
          P^.left    := GP;
          P^.Parent  := LocalRoot;
          GP^.right  := b;
          GP^.Parent:= P;
          if b <> nil then
            B^.Parent:=GP;
          if LocalRoot^.left = GP  then
            LocalRoot^.left := p
          else
            LocalRoot^.Right := p;
        end;
      end;
    end;
  end;

begin
  done         := false;
  Temp := rootPtr;
  GetNewNode(NewNodePtr);
  NewNodePtr^.Priority := NEwPriority;
  repeat
    if NewPriority <= Temp^.Priority
  then begin
      if Temp^.left <> nil then
      begin
        Temp:= Temp^.left;
      end
        else begin
          Temp^.left := NewNodePtr;
          done := true;
        end
    end
      else begin
        if Temp^.right <> nil then begin
          Temp:= Temp^.right;
        end
        else begin
          Temp^.right := NewNodePtr;
          done := true;
        end
      end;
    until done ;
    NewNodePtr^.Parent := Temp;
{see if need rotations to flatten tree }
     Repeat{ determine access path}
        RotateRequired;
     If Rotate>0 then  begin
       DoRotate;
       Rotate :=0;
       end;
     until Rotate = 0;
end; {link}
```

Prog 3: Pascal procedure for inserting event notice into a binary tree using splay rotations.

It is seen that the tree oriented structures perform considerably faster than linear lists for large event sets. Also, we see that there is not a significant difference between the structures for small (i.e. one event) sets. Finally we note that the binary tree structure without rotations was faster than the structure with rotations. Apparently, the unrotated tree remained balanced during our test. Unfortunately we are not able to guarantee that this always is the case. Since the worst case performance without rotations is identical to the performance for linear linked list we recommend the use of rotations.in all cases.

III. PSEUDO RANDOM NUMBER GENERATORS

A. Simple congruential generators

The basic pseudo-random number generator used in almost all simulation programs is the linear congruential generator (LCG) defined as:

X(i+1)= **a** * **X**(i) + **c** MOD **M**

Here the modulus **M** and the multiplier **a** are positive constants and **a** < **M**. The role of the additive constant **c** is to protect against degeneracy by making sure that **X**(i) is never equal to zero. Note that generators using **c** = 0 have the property that the initial seed cannot be equal to zero as a stream of Zeros will be generated if this is the case. This is an annoying feature when using compilers that automatically initialize all integers to zero. .

Setting aside for the moment the issue of the quality of the resulting random number stream, the main problem in implementing an LCG in Pascal is to find a way to deal with the integer overflow that frequently occurs when **a** * **X**(i) is computed. Three approaches are suggested:

1. Hope that your compiler does not recognize integer overflows. Prog 4 gives a TurboPascal implementation of an LCG that relies on this "feature".

2. Define the seed to be of an enumeration type (i.e. [0..65536] and hope that the compile does not check enumeration ranges (this works for several main frame compilers). Some compilers provide commands to disable range checking, check, for example, **"{$rangeeck- "** disables this check for the Microsoft Pascal compiler. The resulting procedure has the same restrictions as those listed above.

3.Use a "portable" computational procedure that avoids overflow. Bratley et al. (1983) gives a procedure for portable generators that avoids integer overflow if a*a < M. This is achieved by breaking the computational procedure into smaller steps each of which involves valid arithmetic. Prog 5 gives a portable generator adapted form L'Eculier (1987)

```
var s:integer;

Function rnnr:real;
 begin
   s := s *3993 +1;
   if s < 0 then s := s + maxint +1;
   rnnr :=s*3.05185e-5;
 end;
```

Prog 4: A linear congruential generator using Turbo Pascal. This generator has a period of 32768 for any initial seed. Other good multipliers are suggested in: Thesen et. al.(1984).

```
var s:integer;

Function Unif:real;
CONST
  A = 162;
  M = 32749;
  Q = 202;{satisfies M=A*Q+r where r <A}
  R = 25;
  SCALEFACTOR = 3.05353e-5;  { 1/M}
var
  k : integer;
begin
  k := s div Q;
  s := a * (s - k*Q) -K*r;
  if s < 0 Then s := s+m;
  unif := s * SCALEFACTOR;
 end;
```

Prog. 5: A "Portable" linear congruential Generator using Turbo Pascal This generator is degenerate for s = 0. Adapted from L'Eculier (1987)

Writing a routine that works is only half the struggle. We also must make sure that the resulting stream of numbers pass reasonable tests for randomness. This is achieved by selecting "good" values of **a, c** and **M.** Among the properties that can be achieved this way are:

- Non-Degeneracy.
- Properties independent of the initial seed
- Passing battery of tests for randomness of sequence
- Passing battery of tests for uniform distribution.

However it should be noted that there are certain intrinsic properties of LCG generators that will always be present in the resulting random number stream. Among these properties are:

- Short cycle (32767) for 16 bit generators.
- Equal intervals between all like numbers.
- x-y plots of output pairs will form lines (with a slope of a).

Thesen et.al.(1984) lists values of **a** that results in reasonable performance for 16 bit generators with **M** = 32768 and **c** =1. Fishman and Moore(1986) presents an exhaustive evaluation of all multipliers for 32 bit computers.

```
Function icombined: integer;
Var
  z,k:Integer;
begin
  k := s1 div 206;
  s1 := 157 * (s1 - k * 206) - k * 21;
  if s1 < 0 then s1 := s1 + 32363;
  k := s2 div 217 ;
  s2 :=146 * ( s2 - k*217) - k * 45;
  if s2 < 0 then s2 := s2 + 31727;
  k := s3 div 222;
  s3 := 142* (s3 - k*22) - k * 133;
  if s3 < 0 then s3 := s3 + 31657;
  z := s1 - s2;
  if z > 706 then z := z - 32362;
  z := z + s3;
  if z < 1 then z := z + 32362;
  iCombined := z ;
end;
```

Prog 6: A long-period, portable generator of uniform integers on 0 32767 (Adapted from L'Eculier(1987)

B. Combined Generators

Most of the weakness listed above can be overcome by combining numbers from several different independent generators. One of the first combined generators was suggested by Knuth (1982), referred to as a shuffle generator, this generator maintains a table of random variates. A random index is drawn, the variate in this position is returned, and it is replaced by drawing from the other random number stream. The period of the resulting stream is equal to the product of the period of the two streams if these periods are relative prime. A draw back of this approach is the fact that a fairly large amount of memory is required to store the required table. Also, fairly substantial initialization is required. A Pascal implementation of a shuffle generator is given in Thesen et.al. (1984)

A more recent combined generator is given by L'Ecuyer (1987). This procedure exploits the facts that:

1) (U1 + U2 + U3) Mod M1 is uniformly; distributed between 0 and M if U1 is a uniform variate between 0 and M, a d U2 and U3 are discrete random variables; and,
2) The period of the combination U1, U2, U3 is the least common multiple of the periods of the three generators.

A sixteen bit implementation of this generator is given in Prog 6. The coefficients used in this implementation were extensively tested, and the resulting performance on spectral tests was shown to be exceptionally good.

C. Constructing floating point variates.

Random variate generation is exceptionally time consuming on micro computes without floating point hardware. This is because at least one floating point division is required

```
Var S1,S2:integer;
```

```
Function uniform:real;
{ Fast generator of Uniforms on 0 -1}
{ From Thesen (1985)               }
var
 k : integer;
 ux: record case integer of
  1:( unif : real );
  2:(ex,m1:byte;
     M4:byte;
     M3:byte;
     M2:byte;
     M5:byte;
     );
   3:(w1,w2,w3:integer);
   end;

 Function Rbyte1:byte;
 begin
   s1 := s1 *3993 +1;
   rbyte1 := s1 shr 8;
   if s1 < 0 then s1 := s1 + maxint +1;
 end;

Function Rbyte2:byte;
 begin
   s2 := s2 *2837 +1;
   rbyte2 :=s2 shr 8;
   if s2 < 0 then s2 := s2 + maxint +1;
 end;

begin
   with ux do begin
     m1 := rbyte1;
     m2 :=rbyte1;
     m3 := rbyte1;
     m4 :=rbyte1;
     m5 :=rbyte1;
     m5 := m5 shr 1;
    ex :=128;
    if m1 < 128 then begin
      m1 := m1 +128;
      ex := 127;
      k := rbyte2;
        while k = 0 do begin
          ex := ex -8;
          k := rbyte2;
       end;
     if k < 128 then begin
       if k >= 64 then ex := ex -1
       else if k >= 32  then ex := ex -2
        else if k >= 16 then ex := ex -3
         else if k >= 8 then ex := ex -4
          else if k>= 4 then ex := ex -5
           else if k>=1 then ex := ex -6
            else ex := ex -7;
        end;
     end;
       Uniform := unif;
   end;
 end;
```

Prog. 7: A Fast Generator of Uniform Variates on 0 -1. This generator uses the (non standard) floating point notation adopted by TurboPascal. A slightly different version is required when using the standard notation. From Thesen (1985).

to scale down a large random integer to the range [0-1]. Thesen(1985) gives a method that avoids this division by independently generating the floating point mantissa and exponent. Different distributions are used for the exponent and mantissa such that the resulting floating point number is in the range [0-1]. The resulting program is given in Prog 7.

The period of the generator given in Prog 7 is unknown, but exceptionally long. The advantages of this generator is its speed and the good empirical properties of the resulting stream of deviates. The weakness of the procedure is the need to do bit-level manipulations and the lack of a strong mathematical theory.

D.Evaluation

A summary of the properties of four different 16 bit generators is given in Table 2. It is seen that Prog 7 is the fastest generator of numbers on [0 - 1] and that the conventional LCG is the fastest generator of integers. The combined generator (Prog. 6) is relatively slow, however it has the dual advantages of portability and good statistical properties.

ALGO-RITHM	Prog	Range	Resol-ution	Period	Time for 10,000
Basic LCG	4	0-32767 0.0-1.0	1 3.1E-5	32767	1.8sec 16.9sec
Portable LCG	5	1-32749 0.0-1.0	1 3.1E-5	32748	2.6sec 18.7sec
Combined	6	1-32362 0.0-1.0	1 3.1E-6	8.1E12	6.3sec 22.6sec
Construction	7	0.0-1.0	4.7E-10	<6.4E8	8.6sec

Table 2: Relative performance of four different pseudorandom number generators for the IBM-PC.

IV. OTHER DISTRIBUTIONS

In this section we present efficient pseudorandom number generators for variates from the exponential, normal and gamma distributions. The reader is referred to Devroe (1986) and Rubinstein(1981) for additional information and for generators of variates from other distriobutions.

A. The exponential distribution

Exponentially distributed random variates are most conveniently generated through the use of inverse transformation:

X := -mean*ln(unif))

where **unif** is a random variate drawn from the uniform distribution on [0 - 1] and **mean** is the mean of the desired exponential distribution. This approach has the advantage of being so simple that a separate procedure may not be required. However, most general purpose **ln** function use a Taylor series expansion with a large number of terms. Each of these terms require a multiplication and a division. The use of the **ln** function may therefor be quite time consuming.

In Figure 5 we suggest an other approach. Based on an idea attributed to Marsaglia by Knuth (1982), we decompose the exponential density function into 13 other density functions, most of which represent distributions that are easier to deal with than the exponential distribution. It is seen that we have approximated the exponential density function using 6 uniform density functions, 6 triangular functions, and, only on the tail, the exponential distribution. The coefficients on Figure 5 were selected such that the maximum error in the resulting linear approximation of the exponential density function is 0.001.

The resulting algorithm is a three stage process:

1. Determine which density function to use:
   A. Select the distribution to be used:
      i .Uniform { P(u)=0.7606 }
      ii. Triangular { P(t)=0.2152 }
      iii, Exponential { P(e)=0.0242 }
   B. Select distribution parameters
2. Generate a random integer using this density function.
3. Convert the integer to a floating point number and scale down as appropriate.

Figure 6 shows the binary search tree that is used in steps 1 and 2 to identify the distribution to be used. Note that we use an integer uniformly distributed on 0 - 32767 rather than a floating point number distributed on 0 - 1. This increases execution speed significantly when micro computers are used.

A Pascal implementation of this procedure is given in Prog 8. The expected level of effort for Prog 8 quite low as the (fast) uniform distribution is used 76.06 % of the time while the triangular function is called 21.5 % of the times. (Two uniform variates are required to generate one triangular variate). The time consuming **ln** function is called only 2.42 % of the time. A comparison between the performance of this algorithm and the conventional inverse transformation algorithm is given in Table 3. It is seen that Prog 8 is seven times faster than the conventional approach.

Method	Time
Inverse Transformation	22 Seconds
Decomposition	3 Seconds

Table 3: Time to generate 1000 exponentially distributed variates on an IBM-PC without an 8087 co-processor.

```
CONST MULT = 3997;
var seed : integer;

{--------------------------------}
 FUNCTION Expo(mean :Real) :Real;
{--------------------------------}

var x : real;

CONST
  PUNIFORM    = 24923;
  PTRIANGULAR = 31975;
  MULT        = 3997;

var{in stead of comparing on 0.0 - 1.0,
     we use ix to compare on 0 -32767}
  ix : integer;

function irand:integer;
begin
  seed :=seed * MULT +1;
  if seed < 0 then seed:=seed+maxint +1;
  irand:=seed;
end;

procedure UseUniform;
const
  P03027 =  7329;  {pr(x<0.3027 =.2236 }
  P06619 = 13401;  {pr(x<0.6619 =.4089 }
  P10965 = 18157;  {pr(x<1.0965 =.5541 }
  P16554 = 21656;  {pr(x<1.6554 =.6609 }
  P24340 = 23892;  {pr(x<2.4340 =.7291 }

begin
  if ix < P06619 then
    if ix < P03027 then
        { x i unif on 0 - .3027}
        { ix is unif on 0 - P03027}
      expo:= irand*9.23795e-6
    else{ x is unif on .3027 - .6619}
        { ix is unif on P03027- P06619 }
      expo:= 0.3027 + irand * 1.09622e-5
        {1.09622e-5 = 0.3592/maxint}
  else
    if ix < P16554 then
      if ix < P10965  then
          { x is unif on  .6619 - 1.0965}
          { ix is unif on P06619 - P10965}
        expo:= 0.6619 + irand * 1.32633e-5
      else{ x is unif on  .10965- 1.6554}
          { ix is unif on P10965 - P16554}
        expo:= 1.0965 + irand * 1.70568e-5
    else
      if ix < P24340 then
          { x is unif on  1.6554 - 2.4340}
          { ix is unif on P16554  - P24340}.
        expo:=1.6554 + irand * 2.37617e-5
        else { x is unif on 2.4340 - 3.7210}
             {ix is unif on P24340-PUNIFORM}
          expo:=2.434  + irand * 3.92773e-5
end;

Procedure UseTriangular;
CONST
  P03027 =  26152;  {pr(x<0.3027=.0375 }
  P06619 =  27384;  {pr(x<0.6619=.0376 }
  P10965 =  28616;  {pr(x<1.0965=.0376 }
  P16554 =  29806;  {pr(x<1.6554=.0363 }
  P24340 =  32014;  {pr(x<2.4340=.0351 }

var itriang,i2:integer;
u: real;

begin
  itriang :=irand;
  i2:=irand;  {min(i1,i2) is triangular}
  if itriang > i2 then
    itriang:=i2;
  if ix < P06619  then
    if ix < P03027 then
        { x is triangular on 0 - .3027}
      expo:= itriang*9.23795e-6
    else
       {x is triangular on .3027-.6619}
      expo:= 0.3027 + itriang
                      *1.09622e-5
         {1.09622e-5 =0.3592/maxint}
  else
    if ix < P16554 then
      if ix < P10965  then
          { x is unif on .6619-1.0965}
          {ix is unif on P06619-P10965}
        expo:=0.6619+itriang*1.32633e-5
        else
           {x is unif on .10965- 1.6554}
           {ix is unif on P10965-P16554}
         expo:=1.0965+itriang*1.70568e-5
      else
       if ix < P24340 then
          {x is unif on 1.6554 - 2.4340}
          {ix is unif on P16554 -P24340}
         expo:=1.6554+itriang*2.37617e-5
       else
         {x is unif on 2.4340 - 3.7210}
         {ix is unif on P24340-PUNIFORM}
         expo:=2.434+itriang*3.92773e-5;
end;

Procedure UseExponential;
var x:real;
begin
  x:=irand*7.38550e-7;
  expo:=-ln((x));
end;

begin
  ix :=irand;
  if ix < PUNIFORM then
     {use unif distr. with p= 0.7603}
    useUniform
  else
    if ix < PTRIANGULAR then
       {use triangular with p= .2155}
      UseTriangular
    else
      UseExponential;
     {get tail from expo with p = 0242}
end;
```

Prog 8: Fast generator of exponentially distributed variates.

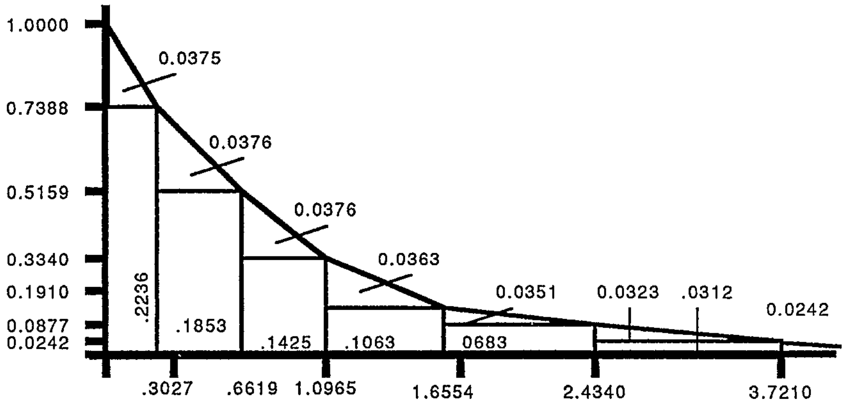


Figure 5: Decomposition of exponential density function into 13 other density functions. Coefficients are selected such that maximum error is less than 0.001

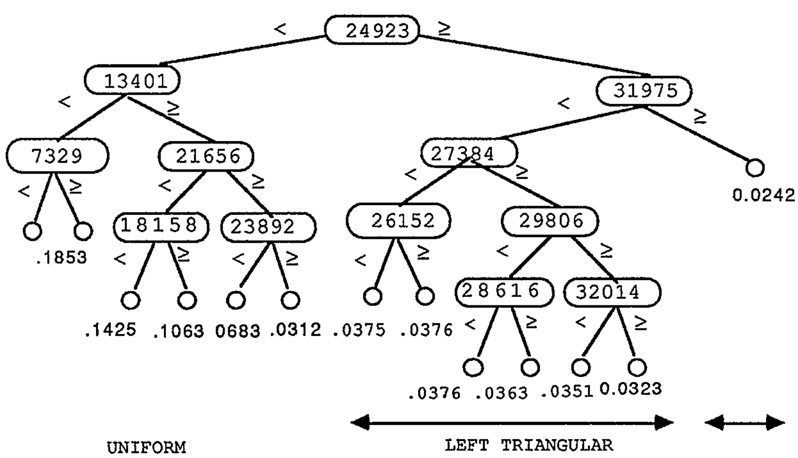


Figure 6: Binary search tree for exponential generator.

B. The normal distribution

Many programmers generate normally distributed random variates by first adding 12 uniformly distributed random variates and then dividing the answer by 12. This approach has the advantages of being simple and of being easy to implement. However it also is computationally slow and it generates numbers from a distribution that is a poor approximation of the normal distribution. Many other approaches to the generation of normal variates are available. Kachitvichyanukul and Lyu (1986) presents an evaluation of 7 such algorithms. A summary of their computational results is given in Table 1.

The three decomposition procedures listed in Table 4 are all quite fast. While the Kinderman & Ramage(76) procedure performed best on the Macintosh, similar tests using different hardware (i.e.IBM-PC) and faster uniform generators (i.e Thesen (1985)) cause the speed advantage of these tree generators to be inverted (Kachitvichyanukul and Lyu(1986)). We therefore hesitate to use speed as the sole criteria for secting on e of these generators. Instead, we consider program size and program complexity. Based on these criteria, we recommend the use of the Kachitvichyanukul(86) procedure. A listing of this procedure is given in Prog 9.

ALGORITHM	REFERENCE	RELATIVE TIME
Decomposition	Kinderman & Ramage(1976)	1.4 sec
Decomposition	Deak (1981)	1.5 sec
Decomposition	Kachitvichyanukul(1986)	1.8 sec
Logistic Majorizing	Tadimakalla(1978)	4.4 sec
Sum of 12 uniforms	Folklore	4.4 sec
Polar Method	Box & Mueller(1958)	4.5 sec
Exponential Majorizing	Tadimakalla(1978)	5.6 sec

Table 4:Time in seconds on an Macintosh to generate 1000 normally distributed variates using different published algorithms.

```
PROCEDURE Normal (VAR ISEED:integer;VAR X:
  real);
{NORMAL GENERATOR by
 VORATAS KACHITVICHYANUKUL
 INDUSTRIAL AND MANAGEMENT ENGINEERING
 THE UNIVERSITY OF IOWA
 modification suggested by Bruce Schmeiser
 December 1986 }
 {-------------------------------------------}

 { Ref: Kachitvichyanukul, V. and Lyu, Jrjung
 On Computer Generation of Normal Random Variables,
 Research Report 84-1, Industrial and management
 Engineering, The Univ.of Iowa}

 CONST
   A = 2.21603587 ;
   P1 = 0.79913208 ;
 VAR
   Accept : boolean ;
   U, V : real ;
 BEGIN
   Accept := FALSE ;
   WHILE NOT Accept DO BEGIN
     U := RAND ( ISEED ) ;
     { REGION 1 TRIANGULAR }
     IF ( U <= P1 ) THEN BEGIN {IF ( U<=P1 )}
       X := A * ( U/P1 - RAND(ISEED) ) ;
       Accept := TRUE ;
     END { IF  ( U <= P1) }
     ELSE BEGIN { ELSE IF ( U > P1 ) }
         { REGION 2 PARALLELOGRAM }
       V := RAND ( ISEED ) ;
       IF (U <=0.97206652) THEN BEGIN{U <= P2}
         X := A * V ;
         V := U/1.59826416 - V + 0.5 ;
       END
         ELSE BEGIN {REGION 3 EXPONENT'L TAIL}
           X := A - LN (V) / A ;
           V := V * 3.0725928 * ( 1.0 - U );
         END;
         BEGIN {FINAL ACCEPT REJECT TEST}
           IF(LN(V) <= ( -X *X*0.5))THEN BEGIN
              { RETURN X OR -X WITH EQUAL PROB}
             IF (RAND(ISEED) <= 0.5 )THEN
               X := -X ;
             Accept := TRUE ;
           END ;
         END ;
       END ;
     END
 END ;
```

Prog 9:: A fast generator of normal variates
Kachitvichyanukul and Lyu(86).

C. The gamma distribution

Schmeiser presents one of the fastest and shortest algorithms for generation of gamma distributed variates with shape parameters greater than one(Schmeiser(80)). A TurboPascal implementation of his procedure is show in Prog 10.. Utilizing a decomposition principle somewhat similar to the one shown in Figure 3 for the exponential distribution, the algorithm first computes the probabilities and ranges for the different regions for the specified values of alpha (shape) and beta (scale) parameters. These are then saved, and reused for consecutive calls. To save setup cost, simulations using several different gamma streams may therefore benefit from the inclusion of independent gamma generators for each stream.

```
TYPE
  GammaDataType = Record
    xLeftTail,XRightTail: real;
    x1,x2,x3,x4,x5:real;
    p1,p2,p3,p4,p5,p6,p7,p8,p9,p10:real;
    F1,f2,f3,f4,f5 : real;
    Alpha, Beta  : real;
  end;
VAR
  GammaData :  GammaDataType;

Function Gamma(NewAlpha,NewBeta:real):real;
VAR
  x,v ,unif1, unif2    :real;
  accept  : boolean;

Procedure MakeGamma;
var
  d : real;
begin
  with GammaData do begin
    alpha := NewAlpha;
    beta := NewBeta;
    x3 := alpha - 1;
    d := sqrt (x3);
    if alpha <= 2. then  begin
       x2 := 0.0;   f1 := 0.0; f2 := 0.0;
       xLeftTail:=-1;
    end
    else begin
       x2 := x3 - d; x1 := x2*(1.-1./d);
       xLeftTail := 1. - x3/x1;
       f1 := exp (x3*ln(x1/x3) +x3-x1);
       f2 := exp (x3* ln(x2/x3)+x3-x2);
    end;
    x4 := x3 + d;
    if d > 0 then  x5 := x4*(1.+1./d);
    xRightTail :=  1 - x3/x5;
    f4 := exp (x3* ln(x4/x3) + x3 - x4);
    f5 := exp (x3* ln(x5/x3) + x3 - x5);
{ calc scaled cum.prob.of each region }
     p1 :=  f2*(x3-x2);
     p2 := p1 + f4*(x4-x3);
     p3 := P2 + f1*(x2-x1);
     p4 := p3 + f5*(x5-x4);
     p5 := p4 + (1.-f2)*(x3-x2);
     p6 := p5 + (1.-f4)*(x4-x3);
     p7 := p6 + (f2-f1)*(x2-x1)*0.5;
     p8 := p7 + (f4-f5)*(x5-x4)*0.5;
     p9 := p8 - f1/xLeftTail;
     p10 :=p9 + f5/xRightTail;
  end;
end;

Procedure AcceptForSure;
begin
    Accept := true;
    with  GammaData do
      if unif1 < p1 then x := x2 + unif1/f2
      else
        if unif1 p2 then x := x3+(unif1-p1)/f4
        else
          if unif1<p3 then x:=x1+(unif1-p2)/f1
          else
            x := x4 + (unif1-p3)/f5
end;

Procedure RectangularRejection;
begin
  unif2 := unif(iseed);
  with  GammaData do
    if unif1 <= P5 then begin
      x := x2 + (x3-x2)*unif2;
      if(unif1-p4)/(p5-p4)<=unif2 then
        accept := true
```

```
      else
        v :=f2 +(unif1 - p4)/(x3-x2)
    end
    else begin
      x := x3 + (x4-x3)*unif2;
      if (p6-unif1 )/(p6-p5)>=unif2 then
          accept := true
      else
        v := f4 + (unif1 - p5)/(x4-x3)
    end;
  end;

Procedure TriangularRejection;
var triangular: real;
begin
  { draw triangular random variabe}
    Triangular := unif (iseed) ;
    Unif2    := unif (iseed) ;
    if Triangular<unif2
    then Triangular:= Unif2;
   with  GammaData do
     if unif1 <= p7 then begin
       x := x1 + ( x2-x1)*triangular;
       v := f1 + 2 * triangular *
            (unif1-p6)/(x2-x1);
        if v <=f2*triangular then
          accept:= true;
     end
     else begin
       x := x5 - triangular*(x5-x4);
       v := f5 + 2.*triangular*
            (unif1-p7)/(x5-x4);
     end;
end;{TriangularRejection}

Procedure Exponential;
begin
  Unif2    := unif (iseed) ;
  with  GammaData do
    if unif1 <= P9 then begin
      unif1 := (p9-unif1)/(p9-p8);
      x := x1 - ln(unif1)/xLeftTail;
      if  x >  0 then
        if (unif2 < (xLeftTail*
           (x1-x)+1 )/unif1)  then
              accept := true;
      v := unif2*f1*unif1
    end
    else begin
      unif1 := (p10-unif1)/(p10-p9);
      x := x5 -  ln(unif1)/xRightTail;
      if (unif2 < (xRightTail*
         (x5-x)+1)/unif1) then
            accept := true
      else
        v := unif2*f5*unif1;
    end;
end;

begin
  with GammaData  do begin
   if NewAlpha <> alpha then
     MakeGamma
   else
     if NewBeta <> beta then makeGamma;
   repeat
     Accept := false;
     unif1 := unif (iseed) * p10;
     if unif1 < P4 then
       AcceptForSure
     else
       if unif1 <P6 then  RectangularRejection
       else
         if unif1 <p8 then TriangularRejection
         else  exponential;
      if not accept then
        if x > 0 then
          if ln(v)<x3*ln(x/x3)+x3-x then accept :=
 true;
     until accept;
     Gamma := beta*x;
   end;
 end;
```

Prog 10: Gamma Generator (Adapted from Schmeiser(1980))

V. SUMMARY

In this paper we have attempted to fill a void in the literature by providing efficient implementations of important algorithms needed in most simulation programs. Needless to say, it has not been possible to provide a comprehensive review of all available algorithms within the page limitations of this paper. Many additional concepts are illustrated in the simulation language **S.Pas** that we developed to evaluate the procedures presented here. A floppy disk containing the source code for this language is available from the author..

REFERENCES

Barnett, Claude C. (1986)," Simulation in Pascal with Micro Passim", *Proceedings of the 1986 Winter Simulation Conference*, J.Wilson, J.Henriksen, S.Roberts(eds), pp141-150.

Box,G.E.P. and M.E.Mueller (1958), "A note on the Generation of Normal Deviates",*Annals of Mathematical Statistics*,Vol.29,No.2,610-611.

Bryant,R.M. (1980). "SIMPAS: A Simulation Language Based on Pascal," *Proceedings of the 1980 Winter Simulation Conference* T.I.Oren, C.M. Shub, P.F. Roth (eds), New York, pp25-40.

Bratley, P., B.L. Fox and L.E.Schrage (1983), *A Guide to Simulation*, Springer-Verlag,1983.

Deak, I. (1981), "An Economical Method for Random Number Generation and a Normal Generator", *Computing* Vol 27, 113-121.

Devroye Luc (1986) *Non-Unif Random Variate Generation*, Springer-Verlag, New York.

Fishman, G.S. and Moore III, L.S.(1986) "An Exhaustive Analysis of Multiplicative Congruential Random Number Generators with Modulus 2^{31}-1,"*SIAM Journal on Scientific and Statistical Computing* Vol 7, No.1, pp 24-45.

Henriksen, J.O., (1977). "An improved events list algorithm." *Proceedings of the 1977 Winter Simulation Conference*, Gaithersburg, MD, 554-557.

Jones, DouglasW. (1986). "An empirical Comparison of Priority-Que and Event set Implementations", *Communications of the ACM*, Vol.29., pp 300-311.

Kachitvichyanukul, V. and Lyu, Jrjung (1986) "On Computer Generation of Normal Random Variables" *Research Report 84-1, Industrial and Management Engineering,* The University of Iowa Revised February 1986.To appear IEEE Transactions on Reliability

Kinderman, A.J. and J.G, Ramage,(1976), "Computer Generation of Normal Random Variables", *Journal of American Statistical Association,* Vol.71, N 893-896.

Kingston, J.H. (1984) "Analysis of Algorithms for the Simulation Event List." *Ph.D. Thesis,* Basser Dept. of Computer Science, Univ of Sydney, Australia.

Knuth, D.E. (1981) *The Art of Computer Programming: Seminumerical Algorithms,* vol 2, Second edition. Addison-Wesley.

L'Ecuyer, Pierre, (1987). "Efficient and Portable Combined Pseudo Random Number Generators", *Department d'Informatique, Universite Laval, DIUL-RR-8612 Revised Edition,* January 1987.

L'Ecuyer, Pierre and Nataly Giroux, (1987) "A Process-Oriented Simulation Package Based On Modula-2," *Proceedings of the 1987 Winter Simulation Conferrence,* A.Thesen, D.Kelton and H.Grant (eds.), pp 165 -174.

Linstrom H. and Skansholm J. (1981) "How to makre your own Simulation System", *Software Practice and Experience,* Vol. 11, pp 629-637.

Livney, Myron (1987) "DeLab - A Simulation Laboratory," *Proceedings of the 1987 Winter Simulation Conferrence,* A.Thesen, D.Kelton and H.Grant (eds.), pp 486-494.

Mallroy, Brian and Mary Lou Soffa (1986), "SIMCAL: The Merger of Simula and Pascal." *Proceedings of the 1986 Winter Simulation Conference,* J.Wilson, J.Henricsen, S.Roberts(eds), pp 397-403.

McCormac, William M. and Robert G. Sargent, (1981). "Analysis of Future Event Set Algorithms for Discrete Event Simulation," *Communications of the ACM* December 1981, Vol.24, No.12,801-812.

O'Keefe, Robert M. and Ruth M. Davies (1986) "Discrete Event Simulation with Pascal" . *Journal of Pascal, Ada and Modula-2.*

Rubinstein, Reuven Y. (1981). *Simulation and the Montecarlo Method,* John Wiley & Sons, New York.

Schmeiser, B.W. and R. Lal (1980) "Squeeze methods for generating gamma variates," *JASA,* Vol.75.

Seila, A. F. (1986), " Discrete Event Simulatiion in Pascal with SIMTOOLS", *Proceedings of the 1986 Winter Simulation Conference,* J.Wilson, J.Henricsen, S.Roberts(eds), pp141-150.

Sleator, Daniel Dominic and Robert Endre Tarjan, (1985). "Self -Adjusting Binary Search Trees",*Journal of the Association for Computer Machinery,* Vol.32, No.3,652-686.

Tadimakalla, P. R. (1978),"Simple Rejection Methods for Smpling from the Normal Distribution", *Proceedings of the 10th Annual Conference of AIDS* ,290-291.

Tadikamalla, P.R. and M.E. Johnson, (1981) ,"A complete guide to gamma variate generation," *American J.Math. and Mgt.Sc,.* Vol 1 213-236.

Thesen, A, Z. Sun and T.J.Wang (1984). "Some Efficient Random Number Generators for Micro-Computers". *In: Proceedings of the 1984 Winter Simulation Confrence.* Sheppard, Pooch and Pegden (eds.) 187-196.

Thesen, A. and Z. Sun (1986) 22 , "The Effect of the Choice of Programming Language on the Performance of of Micro Computer Based Simulation Systems" , in *The Impact of Micro Computers on Operations Research,* K.Hoffman ed. Elsevier Press, 1986..

Thesen, A.(1985) "An Efficient Generator of Uniformly Distributed Variates Between Zero and One," *Simulation.* 44, 17-22.

Uyeno, Dean and W. Vaessen (1980) " PASSIM, A Discrete-Event Simulation Package for Pascal". *Simulation,* Vol 35, No. 6 pp 479.

Vaucher, J.G. and P. Duval (1975). "A Comparison of Simulation Event List Algorithms", *Communications of the ACM,* Vol. 18, No. 4 223-230.

Vaucher, J.G.(1986). Letter to the editor, *Communications of the ACM,* Oct 1986.

AUTHOR'S BIOGRAPHY

ARNE THESEN is a Professor of Industrial Engineering and Computer Sciences at the University of Wisconsin-Madison His current research interests are in the areas of simulation and expert scheduling systems. He is the author of Computer Methods in Operations Research; a text that currently is available in Japanese and Chinese translation.

Arne Thesen
Department of Industrial Engineering
1513 University Ave
Madison WI 53705
(608) 262-3960