Proceedings of the 1986 Winter Simulation Conference
J. Wilson, J. Henriksen, S. Roberts (eds.)

SIMCAL:

THE MERGER OF SIMULA AND PASCAL

Brian Malloy
Computer Science Department
Duquesne University
Pittsburgh, PA 15219

Mary Lou Soffa
Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260

ABSTRACT

Simulation languages, although expressively powerful, usually require a large overhead to learn. In addition, compilers for the languages may be unavailable for a particular computer or may be too expensive. In this work, we develop a process-oriented simulation language, Simcal, based on standard Pascal and extended to directly incorporate the simulation primitives found in Simula. We show that the implementation of Simcal only requires a standard Pascal compiler. The combination of Pascal and Simula provides the simulation modeler with a readily accessible, powerful language for writing simulation programs. The language is currently implemented on a micro computer using a Pascal preprocessor.

1. INTRODUCTION

The use of simulation languages in the design of simulation software supports the widely accepted belief that providing proper programming tools for an application increases programmer productivity. Unfortunately, although many simulation languages have been developed, and continue to be developed, simulation problems are still being coded in general purpose languages. There are a number of possible reasons for this. One difficulty with simulation languages is that, because they are special purpose languages, they may not be available for particular machines. This is especially true for micro computers. Although numerous simulation languages have been implemented on main frame computers, few if any, have been developed for use on a micro. Even if compilers are available for a particular machine, they are frequently expensive, poorly documented and provide very little debugging support.

Another serious obstacle to the use of a simulation language results from the fact that programmers have a tendency to use a programming language that they already know. They are unwilling or unable to invest the time necessary to learn a completely new language. This is a possible reason for the many simulation programs written in Fortran. While Fortran is widely available and well known, it does not include the facilities that simulation languages contain to assist programmers in writing simulation models.

One simulation language that has been used for nearly twenty years is Simula (Birtwistle et al 1980, Lamprect 1983). This language has the desirable feature that it is process-oriented and thus supports the modular design of simulation programs. In a process-oriented language, a sequence of events along with the associated information is grouped **together** as an entity. In Simula, this entity is called a process. The system being modeled can be viewed as a collection of interacting processes.

However, one problem with Simula is the availability and expense of its compilers. Also Simula is based on the language Algol, which is an obsolete language. Users are forced to learn and use Algol as well as the simulation facilities. Thus, programmers are not able to use the data and control structures found in current day programming languages.

This work presents an approach to these problems by developing and implementing a process-driven simulation language, Simcal, based on standard Pascal. This language is extended to directly incorporate simulation primitives designed to have essentially the same syntax and semantics as found in Simula. Therefore a Simcal user, knowledgeable in Pascal, need only consult a Simula reference text for information regarding the syntax and semantics of the simulation primitives. The simulation primitives are directly incorporated into Pascal, meaning that the user is not responsible for adding any calls to system procedures or declaring any extra data structures. This is all handled by a preprocessor for Simcal that takes a Simcal program and translates it into a Pascal program.

We chose Pascal as the base language because it is a current, widely used language. The simulation primitives are based on Simula because it is a well known, process-oriented, simulation language. Simcal was designed as a preprocessor so that it can be used in any environment that has a Pascal compiler and therefore requires no additional software costs. As a preprocessor, it sits on top of the Pascal

compiler, and thus it is not necessary to alter the compiler in any way.

Simcal has been implemented and tested using Turbo Pascal, an inexpensive compiler designed by Borland International running on an IBM PC (Borland 1985). Sample simulation programs written in Simcal have demonstrated that it is efficient in terms of time and space considerations for a micro computer.

Since Pascal is easy to use and readily available at universities, Simcal is an excellent tool for teaching simulation modeling in an academic environment. Due to the fact that most students already know Pascal when they take a course in simulation, the instructor may concentrate on simulation techniques and statistical concepts without having to teach a new language. Also, since Simcal is process-oriented, simulation models are modular in design in keeping with good software development techniques.

In addition to academic applications, Simcal provides easy accessibility to those analysts interested in expressing simulation models using a language designed for simulations. Simcal is especially addressing the needs of those uses who have access to only micro computers.

Previous works have addressed some of the above concerns, and solutions have been proposed. For example, Passim (Uyeno and Vaessen 1980) is a Pascal based, but *event* driven simulation language that provides a package of predefined simulation procedures that can be called in a Pascal program. A similar approach is used in extending Pascal to include coroutines that are then used to implement some simulation primitives (Kritz and Sandmayr 1980). And lastly, techniques are given to extend any language to include simulation primitives similar to those found in Simula (Lindstrom and Skansholm 1981).

In all of these cases, the syntax of the simulation primitives is different from Simula and requires users to learn yet another language. Also, users are responsible for providing additional information due to implementation problems. Restrictions such as no local variables, one form of parameter passing and no multiple instances of processes are also imposed. In Simcal, these restrictions are eliminated and the syntax of the simulation primitives is essentially the same as in Simula. The user does not have to provide any more information than what Simula requires.

2. DESCRIPTION OF SIMCAL

Because Simcal is a process-driven simulation language, there are language facilities to support the creation and manipulation of processes. A system clock and an event list ordered by time are also part of the language. Since it is essential to express the relationships among processes in a simulation, the Simula facility for managing lists is also included.

A process in Simcal is represented by a special "procedure like" block of code called a PROCESS. A process may be acted upon by the simulation primitives ACTIVATE (also REACTIVATE), ACTIVATE AT, PASSIVATE and HOLD. These primitives insert processes into or remove processes from the event list. Processes also may be inserted or removed from user defined lists. Simcal therefore provides primitives INTO, OUT, FIRST, EMPTY, and CARDINAL that may be used to examine or manipulate the user defined lists.

Because the semantics of the simulation primitives in Simcal are basically those defined in Simula, we are only presenting a brief overview here. The reader is referred to a Simula text for a complete description of the primitives (Birtwistle et al 1980, Lamprect 1983).

Processes must be created and scheduled for execution via the event list. To create a process, the primitive CREATE <process name> is used. Note that CREATE is used rather than NEW as is done in Simula, for NEW in Pascal has an entirely different meaning than the NEW of Simula. The CREATE primitive creates a new instance or incarnation of the process named in the argument. A parameter list can be included as part of the process name argument. Currently, the argument is restricted by the implementation to be of type integer. The parameter passing techniques are those of Pascal (i.e., pass by value and by reference). The CREATE primitive initializes the process but the process does not execute until activated with the primitive ACTIVATE. A pointer to the newly created instance is returned by CREATE.

In order to manipulate and reference the newly created instance, a reference variable must be set to point to it (e.g. A:- CREATE PATIENT(I)). Reference variables must be declared prior to their use. This is done using the reserved word REF, with the process name that the variable may reference used as the argument of REF. The reference variable can be used as the argument of ACTIVATE and ACTIVATE AT to activate the process. Also, the reference variable can be used to access variables local to a process via the dot notation. For example, A.X refers to the variable X declared local to the process referred to by reference variable A.

The CREATE primitive is also used to create and initialize user defined lists. When used for this purpose the syntax is L:- CREATE HEAD, where L is a pointer to the head of the list and may subsequently be used to reference

it. The list, initially empty, may be examined and manipulated using list primitives. We will discuss the list primitives later.

Another simulation primitive, ACTIVATE A, inserts process A (already created) as the current element on the event list and thus A becomes the executing process at the current time. Action resumes in A at the point it was last active. ACTIVATE A AT <T> schedules A on the event list to be activated at time T. This primitive can be used to schedule a process that has passivated.

PASSIVATE removes the currently executing process from the event list. The next object in the event chain becomes the current process. The only way that a passivated process can resume execution is by another process activating it. HOLD(T) reschedules the currently executing process to resume T time units from the present time. The next object in the event list becomes the active process. Finally, A.IDLE is a primitive that returns the boolean value true if the procedure A is passivated and returns false otherwise.

In order to facilitate the inspection and manipulation of user defined lists, five primitives are built into the Simcal preprocessor: INTO, OUT, FIRST, EMPTY and CARDINAL. The first primitive, A.INTO(L), inserts process A at the end of list L, where list L has been created previously using the primitive CREATE HEAD. The expression A.OUT removes process A from any list of which it is a member. A process may be in, at most, one list at a time. The primitive L.FIRST returns a pointer to the first process in the list L. The expression L.EMPTY returns true if the list L is empty and false otherwise. Finally, the user may ascertain the number of objects a list contains through the primitive L.CARDINAL, which returns an integer.

3. IMPLEMENTATION OF RUN TIME STATE

A goal of this work was to implement the language Simcal using Pascal as the implementation language without changing the Pascal compiler. In order to do this, there were a number of problems that had to be solved.

The major problem was the implementation of a process by using a Pascal procedure. A process is a persistent procedure (i.e. class or coroutine) in that control can leave the procedure and eventually return with the control state and local variables unchanged between the two events. However, when control leaves a procedure in Pascal, local storage is reclaimed and all information associated with that procedure instance is deleted. Thus, techniques to save and restore both the local storage and control state when execution resumes in a particular instance had to be developed.

Associated with the problem of saving and restoring storage is the associated problem of maintaining multiple entry points within a single procedure. When control returns to a process, the process starts to execute from where it was last active. So, in essence, the Pascal procedure associated with this process must have multiple entry points. To implement a process in Simcal, provision had to be made for multiple entry and exit points from within the Pascal procedure. That is, Simcal must allow control to be passed to another procedure at any point from within and return to the next statement at the proper time. The synchronization of these entries and exits of a procedure must be done carefully in order to match the exits and entrances from the associated process.

Because the processes are manipulated as objects, another problem concerned the implementation of reference variables. A reference variable in Simcal is one that is used to refer to a process. Although Pascal has a pointer capability, a pointer cannot reference a procedure instance.

In addition to the problem of implementing variables that can reference processes, more than one instance of a process may be created and exist simultaneously through the use of the list facility. Clearly, a method for creating and maintaining multiple instances of processes had to be developed.

Further, the problem of the event list and the system clock had to be addressed. An event list is a list of processes each possessing a scheduled time, where the process at the head of the list is the currently executing process. To maintain such a list, a facility for inserting processes into the list and removing them had to be developed. An event manager must be built into the processed code by Simcal in order to allow control to pass to the new current process. Also, since the transfer of control to the process at the head of the event list entails updating system time, the concept of system time had to be incorporated into the solution. Lastly, the relationship among processes expressed through user defined lists had to be implemented.

The solution to the problem of implementing a process by using a Pascal procedure is through data engineering, since control structures can be simulated using data structures. In the case of this particular implementation, the control form of a persistent procedure is implemented through the use of global data structures. Specifically, each creation of a process has two parts:

(1) The usual procedure code and run time instance, and

(2) a record structure used to keep information necessary to implement a process using a Pascal procedure.

Since Pascal provides the capability to reference record structures through pointer variables, the implementation of a variable type that can refer to processes is straightforward in our implementation. Since pointer variables are dynamic structures in Pascal, the facility of multiple instances of processes is accomplished by creating multiple record instances through the standard Pascal system procedure NEW. Each record instance keeps information necessary for the maintenance of its respective process. The record contains fields to restore values of variables and control points, as well as information needed to determine the status (active or passive) of the respective process, pointers to other processes in lists and names of local variables. The record also contains a time stamp field to be used in the maintenance of the event list; this field indicates the value that system time will acquire when this process becomes current.

The EVENT CHAIN is simply a chain of these process record incarnations. The system clock is implemented through a global, real variable, TIME. As these record structures are removed from the event list, the time stamp field maintained in the record structure for each process is used to update the system clock.

4. OVERVIEW OF THE SIMCAL PREPROCESSOR

The preprocessor for Simcal, written in Pascal, takes a source program containing both standard Pascal language constructs and simulation primitives and produces, in **one** pass, a Pascal program that can then be executed on any Pascal compiler. Essentially, the preprocessor performs the following tasks:

(1) Creates record structures to represent processes.

(2) Initializes data structures including the event list and system time.

(3) Inserts Simcal system procedures that implement the actions of the simulation primitives.

(4) Translates Simcal declarations into syntactically correct Pascal code.

(5) Replaces simulation primitives in the source with calls to the system simulation procedures.

(6) Converts the declaration of a PROCESS to the declaration of a Pascal procedure. It also changes the body of the process by inserting code to control multiple entrances and exits of processes.

To accomplish these tasks, the preprocessor contains a procedure that scans the source program for simulation key words that indicate where the source must be altered or insertions made. For example, when the scanner detects the word REF, the preprocessor rewrites this declaration as a pointer variable declaration. When the scanner encounters the word PROCESS in the Simcal source code, the preprocessor is notified, and a persistent procedure is constructed with inserted entry and exit points. To construct this special procedure, a GOTO is inserted as the first statement in the process so that during execution, control passes to a case statement inserted at the end of the procedure. This case statement examines the saved control point in the corresponding record for this process and passes control to the proper control point within the procedure.

The points of control within processes are those places in the source code where one of the three primitives ACTIVATE, PASSIVATE or HOLD is used. These three commands cause a transfer of control to another process. The primitives themselves are replaced with calls to Simcal system procedures.

To demonstrate the Simcal preprocessing of the simulation primitives, consider the implementation of the primitives CREATE and HOLD. CREATE is changed to a call of a Simcal system function that performs the actions of CREATE. Those actions include the creation of an instance of the record structure using the standard Pascal system procedure NEW, which returns a pointer to the record. The fields within the record are initialized and the pointer to the record is returned by the CREATE function call.

To implement HOLD(T), the preprocessor inserts into the source code before the word HOLD, a call to a Simcal system procedure to save local storage. A marker or label must be inserted in the program after the word HOLD so that control can be transferred to this point upon return as well as a call to a procedure to restore local storage. In addition, a call to a system procedure to perform the actions of the primitive HOLD must be inserted by the preprocessor. Essentially this system procedure reschedules the "held" procedure to be activated at TIME + T, where TIME represents present system time. Since the event list is ordered on the record field TIME, this rescheduling requires that the "held" procedure be removed from the event list and reinserted in the proper position. Finally, control is passed to an event manager system procedure that updates system time to reflect the time of the new current process and passes control to that process.

5. CLINIC: A SIMCAL EXAMPLE

As an example of the Simcal language, we present the program in Figure 1. The output from the preprocessor is

given in section 6. The example represents a clinic with three types of processes: doctor, nurse and patient.

```
program clinic;
type name_type = packed array [1..20] of char;
var  REF (head)  q, bloodtest, lounge;
     REF (nurse) tom;   REF(doctor) jackie, bob;
     REF(patient) p;    ctr:integer;

PROCESS patient(n :integer);
{This process represents the nth patient entering the clinic}
begin
  writeln('Patient', n, 'entering clinic at', time);
  if not lounge.EMPTY then ACTIVATE lounge.FIRST
  else PASSIVATE;
  writeln('Patient', n, 'What''s up doc?');    PASSIVATE;
  writeln('Patient', n, 'ouch!!');        INTO(bloodtest);
  if bloodtest.CARDINAL = 1 then ACTIVATE tom;
  PASSIVATE;
  writeln('Patient', n, 'Thank you.');    OUT;
end; {patient}

PROCESS nurse;
{The nurse removes the first patient in line,
 then administers a two minute bloodtest}
var  REF (patient) p;
begin
  while true do begin
    p:- bloodtest.FIRST;
    HOLD(4);          {four minute blood test}
    ACTIVATE p;
    if bloodtest.EMPTY then PASSIVATE;
  end; {while}
end;  {tom}

PROCESS doctor;
{The doctor gives a 2 minute shot}
var  REF (patient) p;
begin
  while true do begin
    OUT; p:- q.FIRST;      p.OUT;
    ACTIVATE p;       HOLD(2);   {2 minute shot}
    ACTIVATE p;
      if q.EMPTY then
      begin  {wait} INTO (lounge);     PASSIVATE
      end; {if}
  end; {while}
end; {doctor}

begin  {main}
  q:- CREATE head;      lounge:- CREATE head;
  bloodtest:- CREATE head;
  tom:- CREATE nurse;
  jackie:- CREATE doctor;     jackie.INTO (lounge);
  bob:- CREATE doctor; bob.INTO (lounge);
  ctr:= 0;
  while time < 5 do begin
    ctr:= ctr + 1;
    p:- CREATE patient(ctr);     p.INTO(q);
    ACTIVATE p;     HOLD(random(6));
  end;
  HOLD(30);
end.
```

Figure 1: The Clinic Example

```
procedure nurse(this_proc :info_ptr);
label 1, 2, 3, 4, 5;
var    p :info_ptr;
procedure save(this_proc :info_ptr);
begin
  this_proc^.vars[1].pval:= p;
end;  {save}
procedure unsave(this_proc :info_ptr);
begin
  p:= this_proc^.vars[1].pval;
end;  {unsave}
begin  {nurse}
  if this_proc^.lab > 2 then
    unsave(this_proc);
  goto 100;  2:
  while true do
  begin
    p:= bloodtest.front^.qnext;
    begin
      save(this_proc);
      this_proc^.lab:= 3;
      HOLD(this_proc, 4); goto 1;  3:
    end;   {four minute blood test}
    begin
      save(this_proc);
      this_proc^.lab:= 4;
      ACTIVATE(this_proc, p); goto 1;  4:
    end;
    if EMPTY(bloodtest) then
    begin
      save(this_proc);
      this_proc^.lab:= 5;
      PASSIVATE(this_proc); goto 1;  5:
    end;
  end; {while}
  finished(this_proc);  goto 1; 100:
  case this_proc^.lab of
    1: goto 1;
    2: goto 2;
    3: goto 3;
    4: goto 4;
    5: goto 5;
  end;  {case} 1:
end;  {nurse}
```

Figure 2: Processed Code for the Nurse in the Clinic Example

The program generates one nurse, two doctors and as many patients as possible within the simulation time period of 5 time units. There is a queue of patients waiting to see a doctor, one to see the nurse and a queue (lounge) for the doctors to wait prior to the arrival of patients.

6. THE PROCESSED CODE

In section 4, the tasks that the preprocessor must perform in processing a Simcal program are enumerated. The code in Figure 2 is the Simcal output for the PROCESS NURSE from the preceding example. Note the replacement of the word PROCESS with the word PROCEDURE to make it compatible with Pascal. Also local procedures SAVE

and UNSAVE have been inserted to save and recover local storage. Additionally, labels have been inserted to mark control points and the case statement has been added at the end of the procedure, as has been discussed.

A call to the system procedure FINISHED has been inserted at the end of the procedure because upon termination of the present process, the next event to occur in the simulation model is the next event in the event list. FINISHED schedules the next event in the event list as the new current process.

7. CONCLUDING REMARKS

Simcal is a compact preprocessor that is implemented in 1414 lines of standard Pascal code using an IBM PC and occupies 42.5K bytes of storage. It has been found to greatly facilitate the implementation of process-oriented simulation models. Several medium sized test cases have been constructed without difficulty. Since Simcal comprises the important primitives of Simula, it is safe to assume that simulation models can be constructed effectively and efficiently with this tool.

Since Simcal was implemented on a micro computer, compilation and execution times are very important, especially in view of the fact that simulation programs generally have long execution times. Since slow running programs on a main frame generally become interminable on a micro computer, this issue is even more germane to this discussion.

Using the clinic example presented earlier as input to Simcal, it was found that 22.68 seconds were required to produce processed code using an ordinary 5 1/4 inch floppy disk on an IBM PC. Furthermore, the compilation and execution of the processed code took an additional 12.5 seconds, using the same machine, for a total of 35.18 seconds. Therefore, approximately one half minute is necessary to process, compile and execute the sample Simcal program using a relatively slow storage device.

To further evaluate Simcal using a faster storage medium, the sample program was processed, compiled and executed using a virtual disk. A virtual disk (also called Ram disk) is a procedure whereby core memory is used to simulate a disk drive. Using a virtual disk (the Six Pack Plus, by AST associates) on an IBM PC, Simcal required 12.19 seconds to process the sample program and 7.3 seconds to compile and execute the processed code, for a total of 19.49 seconds. Thus, compilation and execution times for this Simcal program were reasonably fast. More studies and experience are needed to evaluate Simcal for very large programs. Debugging techniques must also be developed.

In this work we extended Pascal to directly incorporate simulation primitives. The process-oriented primitives are the same as those available in Simula. Therefore, this system will provide wider accessibility to the power of Simula and encourage the design and implementation of simulation models in a simulation language.

Simcal was designed to be used in both a classroom setting to teach simulation as well as for application programs. It has been found to be an effective as well as enjoyable tool to demonstrate process-oriented, simulation programming.

REFERENCES

Birtwistle G., Dahl O.-J., Myhrhaug B., Nygaard K. (1980). *Simula Begin.* Studentlitteratur, Lund.

Borland International Inc. (1985). *Turbo Pascal*, V 3.0, Reference Manual., 4585 Scotts Valley Drive, Scotts Valley, CA 95066, 376 pages.

Franta W.R. (1977). *The Process View of Simulation*, North Holland, New York.

Gordon G, *The application of GPSS V to Discrete System Simulation*, Prentice Hall Inc., Englewood Cliffs, NJ, 389 pages.

Kiviat P.J., Villaneueva, R., Markowitz H.M. (1973). *Simscript II.5 Programming Language*, C.A.C.I. Inc., ed. by Russell E.C., Los Angeles, CA, 384 pages.

Kriz J., Sandmayr H. (1980). Extensions of Pascal by coroutines and its Application to Quasiparallel Programming and Simulation, *Software Practice and Experience*, Vol. 10, pp 773 - 789.

Lafora F., Soffa M.L. (1984). Reverse Execution in a Generalized Control Regime. *Computer Languages*, Vol. 9, No. 3/4, pp 183 - 192.

Lamprect, Gunther (1983), *Introduction to Simula 67*, Friedr. Vieweg & Sohn, Braunschweig/Wiesbaden, 201 pages.

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

Pritsker A.A.B., Pegden C.D. (1979). *Introduction to Simulation and Slam*, John Wiley & Sons, New York, NY, 588 pages.

Saydam T. (1985). Process-Oriented Simulation Languages. *Simuletter*, Vol. 16, N0. 2, pp 8 - 12, April.

Uyeno D., Vaessen W. (1980). Passim: A Discrete-Event Simulation Package for Pascal, *Simulation*, pp 183 - 190.

ACKNOWLEDGMENTS

We wish to acknowledge the support of this work by the National Science Foundation under Grant DCR-8119341.

AUTHORS' BIOGRAPHIES

Brian Malloy received a B.S. degree in mathematics from La Salle University in Philadelphia, a M.Ed. in counselor education and a M.S. in computer science from the University of Pittsburgh. He is currently an assistant professor of computer science at Duquesne University and is working on his Ph.D. in computer science at the University of Pittsburgh. His research interests include software engineering and programming languages.

Brian Malloy
Computer Science Department
Duquesne University
Pittsburgh, PA 15219
(412) 434-6467

Mary Lou Soffa is an associate professor in the Computer Science Department at the University of Pittsburgh. She received a M.S. in mathematics from Ohio State University and a Ph.D. from the University of Pittsburgh in 1977. Her research interests include programming language design and implementation, programming environments and implementation of parallelism. She is a member of ACM, SIGPLAN and SIGSOFT.

Mary Lou Soffa
Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260
(412) 624-6471