A "WAIT UNTIL" ALGORITHM FOR GENERAL PURPOSE SIMULATION LANGUAGES

Jean G. Vaucher
Département d'informatique, Université de Montréal
Case postale 6128
Montréal, Québec, Canada

ABSTRACT

Modern simulation languages such as SIMSCRIPT II and SIMULA 67 are very powerful general purpose languages which contain facilities to handle lists and to schedule events in simulated system time (imperative sequencing statements). These languages do not include some of the useful but more specialized features of previous languages (GPSS, CSL, SOL) especially interrogative sequencing statements such as "SEIZE <facility>" or "WAIT UNTIL <Boolean expression>"; however, the definition capability of the new languages is powerful enough to permit their extension to include the interrogative features.
The addition of some features of GPSS to SIMULA 67 was presented at a previous SIMULATION CONFERENCE. The present paper extends that work by describing an efficient algorithm which adds the "WAIT UNTIL" procedure to SIMULA.

INTRODUCTION

Modern simulation languages such as SIMSCRIPT II [1] and SIMULA 67 [2] are very powerful general purpose languages which contain a relatively small number of special features required for simulation. In common with other modern languages, they have the ability to define complex data structures, allocate memory dynamically and handle lists. The special features are mainly concerned with simulated system time. Each language provides a CLOCK and maintains a list of event notices in chronological order. Each also contains imperative sequencing statements of the form

"SCHEDULE AN event AT timex".

The compilers generate fairly efficient code and the languages can therefore be used for general purpose computing as well as simulation. The power and flexibility of the languages are such that they may be used for any type of simulation.

When comparing the two selected modern languages to some older languages such as GPSS, CSL, SOL etc..., it can be seen that the gain in generality and efficiency has not been attained without some losses. The predefined objects of SOL and GPSS such as facilities, transactions and storages, are not present and interrogative scheduling statements have also disappeared. Interrogative statements are used when it is impossible to predict in advance the system time when an event should take place. An example of such a statement commonly used in GPSS is "SEIZE". If a transaction attempts to SEIZE an occupied facility, its execution is halted until the facility becomes free and it is not possible to know in advance when this event will take place.

A much more general interrogative scheduling statement is found in SOL [3]. This statement has the form

"WAIT UNTIL <boolean expression>"

where the boolean expression is a condition which must be met before execution continues. The condition may refer to any number of state variables.

For certain classes of problems the availability of predefined objects and interrogative statements allows models to be described in a very natural and concise fashion. Programs for the same models written in SIMULA 67 or SIMSCRIPT II will be much longer, complex and error prone although they will tend to use less computer time and memory space.

It might appear that the modern languages are hardly better than FORTRAN, perhaps augmented with some subroutines (GASP), when it comes to simulation. Such hasty judgement overlooks the definition capability of the new languages where it is possible, using the source language, to define the data structures and procedures required to implement the useful features that were frozen in the older languages. Given a sufficiently powerful general purpose language, these extra features can be added to the language so neatly that they appear to be extensions to the compiler.

SIMULA 67 is particularly well suited to this extension philosophy. It contains several elegant mechanisms whereby precompiled routines and object declarations can be added to a user programme. This has already been done at the University of Montreal, adding to SIMULA useful objects of GPSS, such as transaction and facilities, as well as the associated procedures such as SEIZE and RELEASE [4]. The present paper describes the further addition of a WAIT UNTIL feature to SIMULA. The algorithm used to implement this feature is easily described in SIMULA. The simplicity of the description shows quite clearly the sequencing problems inherent in such a powerful feature and permits experimentation with alternative algorithms.

The paper first gives a brief description of some pertinent features of SIMULA 67, then shows some examples of the use of WAIT UNTIL. The implementation is described and the problems arising from the use of TIME as part of the WAIT CONDITION are pointed out. The problems are partially resolved through the use of ALARMS. Finally, there is a discussion of efficiency considerations.

SIMULA 67

SIMULA 67 was developped by Dahl & Nygaard as a successor to ALGOL. In appearance, the language is much like ALGOL although some of the weaker points of its ancestor have been redesigned and other features have been added.

SIMULA 67 has list handling capability and standard list procedures FIRST, SUC and EMPTY will be used in this paper. The genitive or dot notation is used in SIMULA to indicate to which object a procedure should be applied. For example,

"LIST 1.FIRST"

means the FIRST element of LIST 1.

One of the more important lists in the system, the list of event notices or sequencing set (SQS) is maintained automatically by the system and simulated system time is given through the procedure TIME.

One of the important features of SIMULA is the use of PROCESS as a data type. Processes are based on the co-routine concept. Each process has a local instruction counter in addition to its local data and programme. Processes operate in quasi-parallel fashion in roughly the same way as programmes in a multi-programming environment.

Execution of a process may be controlled from another process by scheduling statements such as the following:

1) ACTIVATE process AT timex ;
2) ACTIVATE process DELAY dt ;
3) ACTIVATE process ;

In the first two statements, an active phase for the "process" is scheduled at some time in the future. The last statement is almost equivalent to a procedure call in that "process" is activated immediately, interrupting execution of the activating process; this is called "direct" scheduling.

A process can also schedule itself with the procedures HOLD and PASSIVATE. "HOLD (ts)" causes the process to halt for an interval "ts". PASSIVATE suspends execution of the process for an indefinite period; in this case, the process continues execution only when activated by another process.

Processes can be placed into or removed from lists with the procedures INTO (list) and OUT.

Figure 1 gives a simple but complete SIMULA program using some of the features described. The main item of interest is the definition of CLIENT which is a process. Clients arrive at intervals of 10 minutes, spend a time TS in the system and leave. TS could be any of SIMULA's random generator procedures. There is no contention for resources so that no queues form. N represents the number of customers in the system. The main program activates the first client and, thereafter, each client activates his successor. The main program halts for 1000 minutes giving a duration of 1000 minutes to the simulation.

```
SIMULATION begin
   process class client ;
   begin
       activate new client delay 10 ;
       N := N + 1 ;
       hold (TS) ;
       N := N - 1 ;
   end ;
   integer N ;
   activate new client delay 0 ;
   hold (1000) ;
end ;
```

FIGURE 1 - A complete SIMULA program

It is interesting to note that list handling facilities as well as simulated system time are not strictly part of the basic SIMULA language. These facilities are defined as a standard precompiled extension to the language called SIMULATION. A user indicates to the compiler that he wishes to use this extension by prefixing his program with the name of this extension: hence, the "SIMULATION begin" at the start of the program. In the same way, the standard entities of GPSS and the WAIT UNTIL procedure have been implemented as an extension called GPSSS. To make use of these extra facilities, a user would prefix his program with the name of the extension, "GPSSS".

USE OF "WAIT UNTIL"

This section presents the "WAIT UNTIL" statement as a natural way of expressing complex scheduling rules. Three examples of increasing complexity will be used.

The WAIT UNTIL statement is implemented as a procedure with one parameter which can be termed the "wait condition":

WAIT UNTIL (condition) ;

The condition is a Boolean expression of any degree of complexity. It may even include function calls with side effects. When the procedure is used by a process, further execution is suspended until the condition becomes true. If the condition is true at the outset, there is no wait.

The first example is a modification of the programme of Figure 1. In this example, clients need the use of a facility whose state, free or occupied, is represented by the boolean variable "free 1". Clients wait until, the facility is free then mark it as busy by setting "free 1" equal to false. They use the facility for a time TS then reset the facility to free upon leaving the system. Here the statement, WAIT UNTIL (free 1) is equivalent to a SEIZE of GPSS. The PROCESS definition for this example is shown in Figure 2.

```
process class client ;
begin
    activate new client delay 10 ;
    N := N + 1 ;
    WAIT UNTIL (free 1) ;
    free 1 := false ;
    hold (TS) ;
    free 1 := true ;
    N := N - 1
end ;
```

FIGURE 2 - Use of WAIT UNTIL as SEIZE

In the next example shown in Figure 3, a client must use two facilities one after the other. The facilities are represented by "free 1" and "free 2" and the order of use is immaterial.

```
process class client ;
begin
        ---
        ---
      WAIT UNTIL (free 1 or free 2) ;
      if  free 1 then goto one then two
                 else goto two then one ;
one then two:
      free 1 := false :
      hold (TS1) ;
      free 1 := true ;
      WAIT UNTIL (free 2) ;
      free 2 := false ;
      hold (TS2)
      free 2 := true ;
      goto to over ;
  two then one :
        - - -
        - - -
```

FIGURE 3 - Scheduling dependent on two facilities

Although the use of system time, TIME, as part of the "wait condition" will be shown to present problems, it is often useful to model "balking" where a customer will leave a queue if he has not been served within a certain time.

In the last example, the client wants to use facility I but refuses to wait in line longer than TWAIT unless he sees that he will be the next to be served. In this case, the facility is represented by two integer variables, IN1 and ØUT1, as well as by the state indicator, 'free 1". These variables contain the total number of clients having arrived at the facility and having left the facility. The difference between them indicates the number of clients either being served or waiting for service. The client notes in AHEAD the value of IN1 at the time of arrival; then AHEAD-ØUT1 will give the number of people ahead in the queue.

```
process class client ;
begin
    integer AHEAD ;
    real TØUT ;
       ---
       ---
    AHEAD := IN1 ;
    TØUT := TIME + TWAIT ;
    WAIT UNTIL (free 1 or
         (TIME > TØUT AND AHEAD - ØUT1 > 1)) ;
    ØUT1 := ØUT1 + 1 ;
    if free 1 then goto occupy 1
              else goto exit ;
       - - -
       - - -
end client ;
```

FIGURE 4-Programming of complex balking decision

The WAIT UNTIL statement is clearly a powerful tool for the description of models. It also describes scheduling in a natural manner.

IMPLEMENTATION

The WAIT UNTIL extension has been implemented using the source language facilities of SIMULA and not by modifying the compiler. There are four vital elements to the implementation of WAIT UNTIL:

1) A procedure called WAIT UNTIL to be used by the processes.
2) A list, WAITQ, where processes halted as a result of use of the WAIT UNTIL procedure are kept.
3) a monitor which examines waiting processes and reactivates them at the opportune moment.
4) A global Boolean variable, ACTION, used for communication between waiting processes and the monitor.

The procedure makes use of parameter passing by name. In the procedure, each use of "B" results in re-evaluation of the "wait condition" passed as the parameter. The procedure is given in Figure 5.

```
1  procedure WAIT UNTIL (B) ; name B ;
   boolean B ;
2  begin
3      if B then goto exit ;
4      into (WAITQ) ;
5      if monitor.idle then activate
       monitor after next ev ;
6  loop : passivate ;
7      if not B then goto loop ;
8      out ;
9      action := true ;
10 exit :
11 end wait until ;
```

FIGURE 5 - The WAIT UNTIL procedure

If the wait condition is true initially, the process is not halted and leaves the procedure. Otherwise, the process is placed in WAITQ and passivated. The test to determine if a process may leave WAITQ is done, not by the monitor but by the process (lines 6,7). However, it is the responsibility of the monitor to activate processes when it is possible that the wait condition may be true. When the wait condition is fulfilled, the process leaves WAITQ by using the standard SIMULA procedure OUT (line 8), sets "action" (line 9) to indicate successful exit and carries on execution of its programme. To reduce overhead, the wait monitor is normally idle (passive) and is only activated when processes enter WAITQ (line 5).

The interrogation of the wait conditions could be done continuously, but this is grossly inefficient. Once a process is blocked, it can only continue following a change in system state. With the exception of the system variable TIME, a change of system state may only be caused by an event. Processes in WAITQ need, therefore, only be queried after each event. This periodic examination is controlled by the "wait monitor" whose description is given below:

```
1   process class wait monitor ;
2   begin
3       ref (process) pt ;
4   start : if waitq.empty then passivate ;
5       action := false ;
6       pt :- waitq.first ;
7   loop : if pt== none then goto wait ;
8         activate pt ;
9         if action then goto start ;
10        pt :- pt.suc ;
11        goto loop ;
12  wait : reactivate this wait monitor after
          next ev ;
13        goto start ;
14  end   wait monitor ;
```

FIGURE 6 - The wait monitor

Basically, the monitor goes into action

after each event (line 12) and activates in turn all processes in WAITQ (lines 7-11). The implicit priority is FIFO, new arrivals being placed at the end of WAITQ but a priority scheme could easily be implemented.

If no process is able to leave, the monitor waits until the next event. If, on the other hand, a process leaves WAITQ, this is equivalent to a new event and the monitor passes through WAITQ once again. The monitor carries on testing until no process can advance. It is by testing the boolean "action" that the monitor checks if a process has been able to leave WAITQ.

To reduce overhead, the monitor passivates itself when it finds WAITQ empty. It is reactivated by the first process to enter WAITQ.

PROBLEM OF TIME

The implementation described is very general and works for any possible wait condition with one exception. The exception is the use of the variable TIME. TIME is different from all other system variables in that it changes continuously by itself without posting event notices. This causes some difficulties. Consider the two following statements:

a) WAIT UNTIL (TIME = 15) ;

b) WAIT UNTIL (TIME > 15) ;

and assume that two future events have been scheduled at times 10 and 20 respectively. The processes in WAITQ will only be examined at these two times. The process having executes statement (a) will never be reactivated and the process of statement (b) will carry on at time = 20. This is obviously contrary to the intent of the programmers who intended for their processes to carry on when TIME became equal to 15.

One solution would be to scan WAITQ at fixed time intervals Δt and require that wait conditions involving TIME expressions work with multiples of Δt. This method throws away the advantages gained by using an event-oriented simulation.

The solution that has been adopted is to provide the programmer with dummy events called ALARMS. These are defined in SIMULA by

process class ALARM ; ;

A programmer can now insure correct operation of the "balking" example of Figure 4 by generating an ALARM to trigger an event when the client may wish to leave. The proper procedure is shown below:

```
TOUT := TIME + TWAIT ;
activate new ALARM at TOUT ;
WAIT UNTIL (free 1 or TIME = TOUT) ;
```

DISCUSSION OF EFFICIENCY

Interrogative scheduling as examplified by the WAIT UNTIL statement is a powerful tool, but in the case of simple scheduling decisions, the method is highly inefficient compared to other scheduling algorithms. The inefficiency comes from two main sources.

1) Waiting transactions are placed in a single list irrespective of the wait condition. A change of state in the system leads to examination of all waiting processes.

2) Each waiting process must be tested before knowing if it can continue. A large proportion of the tests will be unsuccessful.

Efficiency is gained by reducing the number of waiting processes that must be examined for a given change in system state. This is achieved by having several wait lists, each corresponding to a distinct wait condition (this is equivalent to the inverted list concept used in structuring data bases). A natural ordering within the list so that only the first or last element need be tested also reduces overhead. If the wait condition for a particular list is specific enough, the repetitive individual testing of each waiting process may be eliminated.

The future events list and imperative sequencing statements found in most simulation languages form an example of these principles. The

The list contains all the processes waiting for the passage of time. Control may be passed to the first element in this list without searching or testing. It is therefore much more efficient to use

```
        "activate this process at TØUT;"
than to use
        "wait until (TIME = TØUT);"
```

When scheduling of a process depends on the state of one variable only, efficiency may be gained if the variable is programmed as a GPSS facility or storage. Efficient implementation in SIMULA of these entities as well as the associated SEIZE and RELEASE procedures has previously been described [4]. It would then be advisable to use

```
    "SEIZE (1);"
rather than
    "wait until (free 1);"
```

It is only in the case of complex scheduling decisions such as presented in the "balking" example that WAIT UNTIL should be used. In such cases, putting a waiting process in several lists, each corresponding to a variable involved in the waiting condition leads to intolerable administrative overhead. If function calls are allowed as part of the wait condition, it may even be difficult to find all the variables involved.

In these cases, the WAIT UNTIL algorithm presented is a useful compromise. It centralizes the scheduling process and makes it systematic. "Ad hoc" methods to give equivalent results are certainly more prone to error and do not appear likely to result in greater efficiency.

CONCLUSIONS

The WAIT UNTIL statement has been shown to be a powerful scheduling tool. It also leads to a natural description of many situations. This feature, present in older languages, is absent from the modern generation of general purpose simulation languages.

WAIT UNTIL has been implemented as a source language extension to SIMULA 67. It could similarly be added to other languages, although the implementation might not be as elegant. Efforts were made to minimize the overhead resulting from use of this feature. The WAIT UNTIL statement is more suitable to complex decision rules. Simple scheduling can be accomplished more efficiently in other ways.

The use of the variable TIME as part of the wait condition was shown to give rise to some problems which could be eliminated by means of dummy event called ALARMS.

ACKNOLEDGEMENTS

This work was supported by the National Research Council of Canada.

REFERENCES

[1] Kiviat P.J., Villanueva R., Markowitz H.M., "The Simscript II programming language", Prentice Hall (1968).

[2] Dahl O.J., Myhrang B., Nygaard K., "SIMULA 67 common base language", Publication S22, Norwegian Computing Centre, Forskningsveien 1B, Oslo 3 Norway.

[3] Dahl O.J., "Discrete event simulation languages" pp. 349-395 in "Programming Languages", F. Genuys (Editor), Academic Press, London (1968).

[4] Vaucher J.G. "Simulation data structures using SIMULA 67", pp. 255-260, 1971 Winter Simulation Conference, New York, Spensored by ACM, AIIE, IEEE, SHARE, SCI, TIMS.