SIMULATION DATA STRUCTURES USING SIMULA 67

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

SUMMARY

This paper describes the approach to the teaching of simulation at the University of Montreal. The language used is the present standard of SIMULA, sometimes referred to as SIMULA 67; however, the modelling approach is inspired from GPSS. This combination of an old and a new language has proved very fruitful.

Little has so far been published on the new standard for SIMULA and it is hoped that the examples in this paper may serve as a painless introduction to simulation programming in this language.

INTRODUCTION

SIMULA [1, 2, 3] is at the forefront of language development. It is a general purpose language with a built-in simulation capability. The language can best be described as an extension of ALGOL with 3 main features:

- "Objects" based on the coroutine concept to allow quasi-parallel processing. An object is a self-contained program with its own local data and actions defined by a generic "class declaration". Examples of objects in simulations would be customers, machines or digital circuits, each of which is capable of independent parallel actions.

- Strong list processing capability and dynamic storage allocation under user control. There is automatic garbage collection.

- The ability to create hierarchies of concepts which makes it easy to generate special purpose problem oriented languages. More specifically, a class declaration prefixed by the name of another class will have the data and actions of that other class added to its own. To a certain extent, the language may be considered as extendable.

The writing of a simulation program in SIMULA is done in two phases. First, the attributes and actions of the objects that will be used are defined through class declarations. In a jobshop simulation, for example, useful objects are jobs and machines. These class declarations are akin to "event routines" in other languages.

Secondly, the required objects are generated and initialised. In the jobshop example, there will be one class declaration for "job" and another for "machine". These declarations will in general be rather complex but generation of an object is fairly simple, for example:

" activate new job at time3 ;"

and as many jobs and machines as required can be generated in this way. The simulation proper is letting these objects interact over simulated time.

In keeping with the philosophy of extendability, the basic SIMULA language contains a minimum of concepts. However, there is in the compiler a predefined class "simulation" which includes an events chain to implement simulated system time. This class also contains useful scheduling and list processing procedures. These procedures and the events chain, called sequencing set (SQS), are expressed in terms of the basic language. The average simulation user would preface his program block with the class name "simulation" to have access to these procedures, but it is important to note that these are not frozen in the language. A programmer is free to ignore the predefined "simulation" class and redefine the implementation of "simulated system time" if he so wishes.

Similarly, an installation interested in "jobshop" simulation could create a "jobshop" class by adding certain concepts and procedures to "simulation". A user could then prefix his program with this new class name "jobshop" and treat SIMULA as a specialized jobshop simulation language[1]. In other words, the first phase of program writing has been eliminated or at least much reduced and the programmer can concentrate on the second phase.

The language therefore allows the user great freedom as to the degree of detail of the modelled objects and the method of passing control between them.

This power, however, has attendant disadvantages:

1) The logic of simulation programs is notoriously complex and unless the programmer uses a systematic approach to program writing, debugging is very difficult. With SIMULA, it is very easy to be clever with a resultant loss of clarity. A good programming methodology is essential.

2) The language does not suggest an approach to modelling. It was found, when first teaching SIMULA, that students had a tendency to model in too great detail. As a result, they had difficulty in abstracting concepts from one model to adapt them to another.

GPSS[4] shows quite a different approach to the design of a simulation language. GPSS forces a "world view" on its user by providing him with predefined objects. GPSS views systems in terms of transactions flowing from one work area to another. The objects of GPSS are transactions, facilities, storages and logic gates. The transaction is the only object whose behaviour the user can describe. In fact, the transaction description is the GPSS program. The other objects are standard and their behaviour predefined. The language is restrictive and it does not have the power of SIMSCRIPT or SIMULA, but the wide use and popularity of GPSS shows that its choice of basic objects is well founded and quite useful for a large class of problems.

GPSS is especially helpful for the beginner on just those two points noted as disadvantages for SIMULA. Model building is simplified by the presence of the predefined objects which can be used as building blocks. All decisions for scheduling movements through the system are localised in the transaction definition. The other objects are completely passive in this style of modelling. The model logic is therefore easy to debug. Further, GPSS programs for queue-type simulations, for which they are well suited, are very concise.

It is therefore advantageous to combine the two languages. SIMULA definitions for "facilities" and "storages" will be given as well as the corresponding "entering" and "leaving" procedures. The "transaction" is also defined. By prefixing his program with these definitions, a SIMULA user can program in a way closely resembling GPSS. Essentially, programming is adding to the "transaction" definition.

The concepts of "facilities" and "storages" to represent resources are so fundamental to scheduling or queuing problems that they may be considered basic simulation data structures. These structures are very close to the concept of "semaphore" introduced by Dijkstra to synchronise loosely connected parallel processes[5]. The "entering" and "leaving" procedures correspond to his "P" and "V" operations. In this way, the old concepts of GPSS are finding new life in systems programming.

The paper will refer to the SIMULA concept of "object" rather than "data structure". The two are close although the SIMULA object includes relevant actions as well as the data structure itself.

DEFINITION OF "FACILITY" AND "TRANSACTION"

The facility is a resource that can be used by only one transaction at one time. It is in one of two states, busy or not busy. Each facility has an input queue where arriving transactions wait when the facility if occupied by another transaction. In this first example, the queue discipline is FIFO. Transactions have two special procedures. With the "seize" procedure, the transaction tries to occupy the facility. If the latter is busy, the transaction will stop execution of its program and place itself at the end of the input queue. With "release" a transaction frees a facility and allows the next transaction on the queue to occupy the facility and resume its program.

The following SIMULA block describes a class "GPSS" which defines transactions and facilities. The underlined words are basic SIMULA symbols and the capitalized words are either system variables or procedures already defined in the system class "SIMULATION".

Program 1

```
1         SIMULATION class  gpss;
2         begin
3                 ref (facility) array
                  station [1:100]
4         class facility ;
5         begin
6                 ref (HEAD) inq ;
7                 ref (transaction) occupier ;
8                 boolean procedure busy ;
9                    busy : = occupier =/=  none ;
10      inq :- new HEAD ;
11  end facility definition ;
12        PROCESS class transaction ;
13        begin
14              real time mark, priority ;
15          procedure seize (n) ; integer n ;
16             begin
                 if station [n] = = none then
                 station [n] :- new facility ;
17               if station [n] . busy then
                 begin
18                  INTO (station [n] . inq) ;
                    PASSIVATE ;
19                  OUT ;
                 end ;
20               station [n] . occupier :-
                 this transaction ;
21             end seize ;
22      procedure release (n) ; integer n ;
23       begin
24         inspect station [n] when facility do
25         begin
26           if occupier =/= this transaction
             then ERROR ;
27           if inq. EMPTY then occupier :- none
28           else activate inq. FIRST delay 0 ;
29         end
30       end release ;
31       time mark :=  TIME ;
32  end  transaction definition ;
```

33 end definition of class gpss ;

line 1 By prefixing the class definition with the name of the system class SIMULATION, all the scheduling and list processing procedures of that class are made available to the class GPSS .

line 3 A ref type variable is a name or pointer to an object. The declaration also limits the use of the name to objects of a certain class; in this case, "station [n]" can only refer to a facility. Neither the class definition nor the ref declaration create the objects in question. Objects of a certain class, for example "facilities" will have to be created by a statement of the form:
station [17] :- new facility ;
Thereafter, the new facility can be accessed or referenced through its name "station [17]" .
The declaration of line 3 limits the simulation arbitrarily to 100 facilities.

lines 6,7 These are the attributes of a facility. "inq" is the name of the input queue for waiting transactions. In SIMULA, the list processing capabilities are based on two-way circular lists where a special marker, a "HEAD" object, serves as both the starting and end point of the list. A queue is defined as a pointer to a "HEAD" object.
"occupier" is used to contain the name of the transaction which has seized the facility.

line 8 A variable could have been used to indicate wether or not the facility is busy, but this information can easily be obtained by the procedure shown which tests if an "occupier" exists.

line 10 This initialisation is executed at the creation of each facility. An empty list is created and its address is placed in "inq". At creation, ref variables are = none and the procedure "busy" would therefore return a "false" value.

line 12 PROCESS is a system class. Roughly, it enables an object to be scheduled and event notices for the object can be inserted into the Sequencing Set. Contrary to simple objects, the program part of a PROCESS object is not execute at creation but only when the object is first activated.

line 14 These have been found to be useful attributes for transactions. The first is used to time the transaction through parts of a system. "Priority" is not used in this example; the next section gives a more complex procedure "seize" which uses this variable to give different queuing disciplines.

line 16 The first time a transaction tries to seize a facility that has not been created explicitly, it is automatically created.

line 17 If the facility is free, the transaction does not enter the queue but jumps to line 20 to occupy the facility. Note the use here of the "genetive" or dot notation to access a particular facility's attributes: "station [2] . inq" means "station [2]'s inq" or "the inq belonging to station [2]" .

line 18 INTO is a system procedure which places the transaction at the end of the queue passed as a parameter (FIFO). The procedure PASSIVATE stops the execution of the transaction program at this point until such time as it is reactivated by some other object (see line 28).

line 19 As soon as it is activated, the transaction executes the procedure OUT which removes it from the input queue.

line 20 The transaction places its own name in the occupier variable.

line 24 When it is required to have access to many of an object's attributes, the dot notation becomes tedious. The "inspect" statement provides a useful shortcut. In the statement or block following the do any reference to a facility attribute is assumed to refer to the attributes of the station [n] facility and the dot notation is superflous.

line 26 Here we guard against the possible logic error where a transaction tries to release a facility it never seized. ERROR is not a system procedure; its implementation is left free. ERROR could possibly print a message and jump to the end of the procedure so that no action would be taken.

line 27 If the queue is empty, the facility is marked as not busy.

line 28 The first transaction in the input queue is reactivated. The delay 0 clause enables the leaving transaction time to complete its actions before control is passed to the waiting transaction.

line 31 TIME is a system procedure which gives the simulated system time. "Time mark" is therefore initialised to the time at which the transaction enters the system. All integer and real variables are automatically set to zero at block entry and the priority variable does not need explicit initialisation.

QUEUING EXAMPLE

To show the similarity between a GPSS program and a SIMULA program using the previous definitions, a single server FIFO queue is considered. A typical fragment of a program is given below:

Program 2

```
    GENERATE     5,0          GPSS EXAMPLE
    MARK
    SEIZE        1
    ADVANCE      4,1
    RELEASE      1
    - - - - -
    - - - - -
    TABULATE     1
1   TABLE        M1,5,5,10
    TERMINATE    1
    START        200
```

Transactions arrive in the system every 5 minutes. Arriving transactions are time stamped then seek to enter facility 1. Service time in facility 1 is uniformly distributed between 3 and 5 minutes. The dashes indicate that transactions then go through other parts of the system. The time through the system is tabulated then the transactions leave the system. The simulation is ended after 200 transactions.

The complete equivalent SIMULA program is:

Program 3

```
1   gpss begin
2   transaction class customer ;
3   begin
4        activate new customer delay 5 ;
5        time mark  :=  TIME ;
6        seize (1) ;
7        HOLD (UNIFORM (3,5,u)) ;
8        release (1) ;
         - - - - - - - -
         - - - - - - - -
         - - - - - - - -
9        OUTFIX (TIME - time mark, 2,10) ;
10  end  customer description ;
11
12  comment now the initialisation ;
13  integer u ;
14  u := 54321 ;
15
16  activate new customer delay 0 ;
17  HOLD (1000) ;
18  end of the simulation
```

The class declaration for the "customer" object adds scheduling statements to the predefined "transaction". This customer declaration is the exact equivalent to the GPSS program. Line 4 is equivalent to the generate block; upon arrival in the system each transaction schedules the arrival of its successor. Line 5 is the same as the "mark" block. This line is not strictly necessary since "time mark" is initialised to time of entry. HOLD stops execution of the program for the indicated amount of simulated time. UNIFORM is a standard function which returns a random value uniformly distributed between 3 and 5. "U" is the random number seed. The time through the system is also printed.

The rest of the program deals with initialisation and control of the simulation. For more realistic, larger models, only the customer declaration would increase in proportion to the complexity. The initialisation would stay almost identical. Line 16 schedules the exogenous event which starts the simulation and in the next line the main program relinquishes control for the duration of the simulation. In a more realistic model statistics would be collected and printed out at this point.

The random number seed "U" is declared and initialised in lines 13 and 14. To clean up the program these two lines could be inserted in the GPSS definition instead.

In spite of some minor differences, the structure of the two programs is very similar.

PRIORITY QUEUING

In queuing models, it is frequently necessary to experiment with a wide variety of queuing disciplines. In the simple "facility" described so far the discipline in the input queue is FIFO. Entering transactions are placed at the end of the queue and when the facility is free it selects the next occupier from the beginning of the same queue. A wide range of disciplines can be introduced by modifying the "seize" procedure so that the incoming transaction is placed in the "inq" according to some priority parameter. The following procedure places transactions in a queue in ascending order of priority; in case of identical priorities, the ordering is FIFO is the priority is positive and LIFO if the priority is negative.

Program 4

```
    procedure priority into (queue) ;
    ref (HEAD) queue ;
    begin
       ref (transaction) pt ;
       pt :- queue.FIRST;
    loop: if pt = = none then INTO (queue)
       else if priority < pt.priority
       then PRECEDE (pt)
       else if priority = pt.priority
       and priority < 0
            then PRECEDE (pt)
       else begin
            pt :- pt.SUC ; goto  loop ;
            end ;
   end proc priority into ;
```

To implement priority queuing, the procedure should be placed in the transaction definition and line 18 (program 1) of the "seize" procedure should have the "INTO" procedure replaced by the newly defined "priority into".

As an example of priority queuing, consider a transaction which passes successively through 4 facilities. The service times in each are "ts1", "ts2", "ts3", and "ts4" respectively. Each transaction has two local variables: "code" and "delivery date". The variable "code" indicates the importance or priority of the transaction (code=1 has priority over code=2). The discipline at each facility is as follows:

```
station 1 : LIFO
station 2 : shortest service time-first out
station 3 : low code-first out (priority)
station 4 : earliest deadline - first out
```

The transaction program for this example is:

Program 5

```
- - -
- - -
priority := -TIME ; (LIFO)
seize     (1) ;
HOLD      (ts1) ;
release   (1)

priority := ts2 ;   (priority to short ser-
                         vice time)
seize     (2) ;
HOLD      (ts2) ;
release   (2)   ;

priority := code ;  (priority to lowest co-
                                 de)
seize     (3) ;
HOLD      (ts3) ;
release   (3)   ;

priority := delivery data ; (priority to
             earliest delivery date)
seize     (4)
HOLD      (ts4) ;
release   (4)   ;
- - -
- - -
```

In cases where the relative transaction priority is liable to change while it is waiting in the queue, the above scheme for priority queueing is no longer valid. In this case, the old "seize" procedure would be kept and it would be the "release" procedure that would be modified. More specifically, the procedure FIRST in line 28 should be replaced by a procedure "priority first" to select the transaction with the lowest value of the priority variable. Using the full power of SIMULA, "priority" could also be defined as a virtual procedure allowing dynamic recalculation of the priority.

DEFINITION OF "STORAGE"

A storage is a resource that can be subdivided. Parts of it can be allocated to different transactions; each transaction having exclusive use of its own part. The main characteristic of a storage is its maximum capacity. A storage can be used to represent a group of identical facilities. Examples of storages are: computer memory allocated to several programs, space in a warehouse or salesmen in a shop. Transactions request use of storage through an "enter" procedure where they indicate the number of units of storage required. Units are returned to the storage through the "leave" procedure. There is no check wether a transaction returns the same number of units that it took or even if it previously removed any units at all. The only verifications are: 1) a transaction cannot request more than the maximum capacity of the storage, and 2) returned units cannot increase the available space over the maximum capacity.

There are many ways to implement a "storage" with "enter" and "leave". The following declarations show a possible version whose implicit scheduling decisions will be discussed later.

Program 6

```
ref (storage) array box [1:100] ;
class storage (capacity) ; integer capacity ;
begin
    ref (HEAD) inq ;
    integer available space ;
  procedure check inq ;
  begin
       ref (transaction) client, next client ;
       client :- inq. FIRST ;
  loop: if client = =none or available space = 0
       then goto fin ;
       next client :- client . SUC ;
       activate client ;
       client :- next client ;
       goto loop ;
  fin : end of check inq ;
   inq :- new HEAD
   available space := capacity ;
end storage ;
```

The maximum capacity is specified as a parameter when creating the storage. For example, to model a parking lot with space for 100 cars, the storage would be created as follows:

```
parking :- new storage (100) ;
```

The following procedures should be added to the transaction definition of PROGRAM 1. Both the "enter" and the "leave" procedures specify the identity, "n", of the storage and the number of units of storage required or released.

```
procedure enter (n, units required) ;
integer n, units required ;
inspect box [n] when storage do
begin
     if units required > capacity then ERROR ;
     INTO (inq) ;
test : if units required ≤ available space then
       goto exit ;
     PASSIVATE ;
     goto test ;
exit: OUT ;
     available space :=  available space -
     units required ;
     HOLD (0) ;
end enter ;
procedure leave (n, units released) ;
integer n, units required ;
     inspect box [n] when storage do
     begin
        available space := available space +
        units released ;
        if available space > capacity then ERROR ;
        check inq ;
     end leave ;
```

To see the actions of the different procedures, consider a transaction trying to enter a full storage. It puts itself at the end of the input queue then tests if space is available. In this case, the test is false and the transaction is passivated. Later, whenever the transaction is activated, it will perform the same test. When a transaction leaves the storage, it returns some units to "available space" and calls "check inq". This procedure scans through the input queue on a FIFO basis, activating each transaction in turn until either it reaches the end of the queue or the available space becomes equal to zero.

The scheduling decisions implicit in the implementation are:

1) The input queue discipline is FIFO. One could choose to have priority queueing as for the facility.

2) Blocking is not allowed. That is, when a small amount of space is returned to the storage, all waiting transactions are given a chance to enter. A transaction at the start of the queue which requires more space than is available does not prevent a succeeding transaction requiring less space from entering. "Check inq" could be modified to implement blocking by stopping the scan of the input queue as soon as a transaction was too large to be accomodated.

3) At present, an entering transaction tests for available space independently of the number of waiting transactions. If "blocking" were implemented, this could only be tolerated with an empty queue.

CONCLUSIONS

Conclusions from this work can be drawn on several levels:

1) It has been shown how SIMULA can imitate GPSS. By prefacing his program with the definitions given here, a user can program a suitable model with the same ease and brevity as in GPSS. The use of well defined objects eases model building and later debugging. The approach has proved quite successful in the introductory teaching of simulation with SIMULA.

2) SIMULA is not limited to GPSS type models. The power of the language has been shown in the definitions of GPSS objects. The structure of these objects is not frozen as in GPSS and some alternative algorithms for the implementation of "storage" have been suggested. SIMULA is also suited to other types of models. The language has been applied to biological epidemic simulations^2 and at Montreal, SIMULA has been used with success to simulate both PERT and digital networks.

3) Good programming in SIMULA leads to a certain formalisation of generally useful data structures for simulation: "facilities" and "storages" in the present paper and "nodes" and "arcs" in our other simulations of networks. SIMULA appears to be an ideal language with which to study these structures and find efficient ways to implement them. The SIMULA definition of "storage" points out the various alternatives in scheduling entry and indicates the complexity of what appears at first to be a relatively simple concept.

4) Finally, the ease of definition in SIMULA of complex data structures and their associated routines suggests that simulation algorithms or the structure of models be described in SIMULA just as computation algorithms are described in ALGOL.

REFERENCES

1) O.J. Dahl, B. Myhrhaug and K. Nygaard, "Some Features of the Simula 67 Language", 2nd Conference on Applications of Simulation, New York, December 1968.

2) O.J. Dahl and K. Nygaard, "SIMULA - an ALGOL Based Simulation Language", Comm. of the ACM, Vol.9, September 1966.

3) O.J. Dahl, B. Myhrhaug and K. Nygaard, "SIMULA 67 Common Base Language", Norwegian Computing Centre, Oslo, May 1968.

4) I.B.M., "General Purpose Simulation System /360, User's manual", I.B.M. Publication H20-0326, 1968.

5) E.W. Dijkstra, "Co-operating Sequential Processes", in "Programming Languages", Genuys (Ed.), Academic Press, 1968.