? 1977

Discrete Event Simulation Based on Communicating Sequential Processes

W. H. Kaubisch and C. A. R. Hoare

Department of Computer Science*
The Queen's University, Belfast, N. Ireland.

This paper suggests a minimal set of primitive concepts required in the construction of algorithms for discrete event simulation. The basic concept is the communicating sequential process [CSP]; however, this is reinterpreted as a quasiparallel process, operating in simulated time. The most important features of simulation are shown to be implementable as communicating processes, and a nontrivial example of a simulation algorithm is given.

Key words and phrases: Programming Languages, Primitive Concepts, Discrete Event Simulation, Quasiparallel Processing, Communicating Sequential Processes.

CR Categories: 4.22, 3.65

---

*This research was supported by a grant and a senior fellowship from the Science Research Council of Great Britain.

1. INTRODUCTION.

Research into programming languages has produced a wide variety of proposed designs. Each design attempts to improve upon its rivals, often by introducing additional "powerful" features, perhaps oriented towards a particular application area. As a result, some of the languages have been quite complicated to implement and even to understand; but many programmers have taken this as a challenge to their skill and ingenuity.

This paper takes exactly the opposite approach; it attempts to remove as many features as possible from previously proposed languages, and replace them by the barest minimum set of primitive structures, which are adequate for the intended purposes. It suggests that, even in a specialised application area such as discrete event simulation, most requirements can be met by a few general-purpose features.

The paper starts with a brief survey of the main requirements of a programming language designed for discrete event simulation, namely, resources, processes, simulated time, queues, statistics, and random numbers. These are illustrated by features and examples of the use of existing simulation languages, GPSS [5], SIMULA 67 [2], and SIMONE [4].

The next following section introduces the concept of a communicating sequential process [1], which is interpreted as operating in simulated time instead of real time. The proposed notation is described by means of annotated examples, since a fuller and more formal description is already available [1]; however, as an experiment, this earlier language has been extended by permitting output commands to appear in guards.

In section 4, a series of examples illustrate how this language can be used to implement all the basic requirements of simulation, as surveyed in section 2. It culminates in a simple but complete simulation algorithm, the machine shop [3].

2. THE REQUIREMENTS OF DISCRETE EVENT SIMULATION.

This section surveys the requirements of discrete event simulation, and the features which have been added to programming languages to adapt them for this purpose.

2.1 RESOURCES.

Among its definitions for "resource", the Oxford English Dictionary gives:

1. A means of supplying some want or deficiency; a stock or reserve one may draw upon when necessary.
2. An action or procedure to which one may have recourse.
3. The capability of adapting means to ends ...

The first definition refers to stocks and reserves, which we may interpret as being supplies of inanimate objects or materials, for example, the components or metal used in a machine shop. The second definition refers to actions and procedures executed by agents, which are of interest not because of their physical substance but because they accomplish some desired result; the machines and machine operators in the machine shop are examples of this type of agent. According to the third definition, we may say that the machine shop itself is a resource, because of its ability to adapt the means at its disposal (i.e. the metal, the machines and the operators) to meet a given set of orders.

Simulation models and languages must be capable of representing such widely varying types of resource. In the simplest cases, resources may be represented by simple variables; but more generally their representation will require the use of structures, e.g., the PASCAL [2] RECORD structure, or even structures with associated rules of access, such as the SIMULA 67 CLASS [3],or module of MODULA [6], or by the MONITOR of SIMONE [4].

As an example, in the case of the machine shop, the stock of metal may be represented as an integer giving the amount of stock on hand e.g.

```
METALONHAND : INTEGER;
```

The simplest representation of a (single resource) machine is as a boolean variable, indicating whether that machine is in use:

```
MACHINEINUSE : BOOLEAN;

MACHINEINUSE := FALSE;
```

Going on to a more elaborate example, consider a group of machines, each of which is capable of doing the same job, but each of which has a different running cost. Each machine could now be represented as a RECORD:

```
MACHINE = RECORD
            RUNNING COST : INTEGER;
            INUSE ; BOOLEAN
          END;
```

and a group of ten of these machines can be represented as an array:

```
MACHINEGROUP ; ARRAY [1..10] OF MACHINE;
```

Finally, assume that the machine group has a foreman whose job it is to decide about the usage of his machines. When a customer wishes to use a machine he must first ask the foreman for permission, and when he has finished, he must inform the foreman of this fact. In the language GPSS, a resource like the foreman would be represented by the built-in STORAGE feature. In other languages, it must be programmed explicitly - for example, as a monitor in SIMONE:

```
MONITOR FOREMAN;
  ... declaration of local variables ...;
    PROCEDURE REQUEST;
        ... body of request ...;
    PROCEDURE RELEASE;
        ... body of release ...;
    initialisation of local variables ...
END
```

Here, the procedures REQUEST and RELEASE may be called from outside the monitor by qualified calls:

(1) FOREMAN.REQUEST;

which acquires a machine from the foreman, possibly after some delay.

and (2) FOREMAN.RELEASE;

which returns a machine to the foreman for reallocation.

There is a qualitative difference in complexity between the foreman and the previous examples. One could write ever more complex records to represent ever more complex inanimate resources; but they remain inanimate, and the manner and sequence of access to them is determined solely or mainly by the accessing program. However, a monitor or a class has the capability of apparently autonomous behaviour. The foreman, in his efforts to grant a request, may himself initiate requests for other resources; for example, he may make choices about maintenance schedules, may call for a mechanic, etc. Hence, one request may set off a chain of other events, and the interaction between the resources may become quite complex. We shall see later that the representation of such a complex resource can take advantage of the full generality of a communicating sequential process.

2.2 PROCESSES.

A process is an independent action or series of actions leading to the realisation of some result. Processes can interact with each other when they compete for resources or communicate ~~with one another~~. Apart from these interactions, processes are independent of each other; and in particular they make independent progress in time, ~~and~~ in that a number of processes will be executed concurrently (in parallel). In a simulation, this concurrency is usually implemented by interleaving actions from all the active processes (also in quasiparallel).

In a machine shop simulation, an example of a process would be an order which flows through the shop. Each order specifies the series of steps required to produce the desired product; each order is independent of other orders except insofar as it uses a common set of resources (machines). In SIMONE [5] an order may be represented as a process:

```
PROCESS ORDER;
BEGIN  ... declaration and initialisation of local variables ...;
  FOR I := 1 TO NUMBEROFSTEPS DO
 BEGIN request,use and release the machine required
       for this step ...
  END
END;
```

The process-like quality of the ORDER is obvious, because it is not "used" by any other process. We have already seen that type 2 and 3 resources also exhibit process-like characteristics relative to the resources they manipulate, e.g., from the point of view of ORDER, the FOREMAN is a resource, yet the FOREMAN itself behaves as a process relative to the machines, materials and mechanics which it schedules. Thus it appears that the processes and resources of a simulation algorithm display a multilevel tree organisation. At the bottom level are the type 1 inanimate resources, at the top level are the pure processes; and in between are the type 2 and 3 resources. Looking from the top down, every structure looks like a resource; looking from the bottom upward, they look like processes. The process and resource concepts are relative rather than absolute. Even the processes (orders) at the top level would appear to be resources if we were to add another level to the tree, for example, customers who originate and cancel orders. It is this insight which will enable us to represent both processes and resources of types 2 and 3 by a single primitive program structure, the communicating sequential process.

2.3 TIME.

A simulation algorithm describes not only the static elements and relations of the system being modelled, but also the dynamic behaviour and interactions of the processes and resources as they evolve in time. But the passage of time must itself be simulated as the algorithm is executed; and each process which engages in an activity which is intended to take an appreciable amount of time must specify its duration explicitly.

In SIMULA 67 and SIMONE, the current value of simulated time may be discovered by a call on the parameterless function TIME, for example

```
IF TIME < FIVEOCLOCK THEN ...
```

When a process is to engage in an independent activity which will last D units of time, this is indicated in the program by a call on the standard procedure HOLD(D). If the value of TIME before this call is T, then the value of TIME after the call will be T+D. Thus the effect of HOLD is simply to suspend the calling process until the elapse of the specified duration of simulated time. For example, the loop of the ORDER process (of the previous section) may be given in greater detail:

```
FOR I := 1 TO NUMBEROFSTEPS DO
BEGIN FOREMAN.REQUEST;
      HOLD(USAGETIME);
      FOREMAN.RELEASE;
END.
```

Here, we are not interested in the details of what the order does with the machine; we are interested only in the fact that its usage of the machine continues during the specified interval in model time, during which the process engages in no other interaction or change of state.

Simulated time must not be confused with the real time taken by a computer in execution of the commands of a simulation program. If a program does not contain any HOLD operation, the entire program would be executed at the same instant of simulated time, though it would certainly take some real execution time on a computer. Conversely, when every process of a program is engages in a HOLD, they are using no computer time; but on each such

occasion, the implementation of the simulation language steps on the value of simulated time to the earliest value which would permit a process to resume execution after its HOLD. Thus it may be said that movement in simulated time takes no execution time, and vice versa.

2.4 QUEUES.

A process which requires to use a resource will usually have to wait if that resource is busy. If several processes have to wait for the same resource, they will have to form some kind of queue. When the resource becomes free, a choice must be made between the waiting processes, on the basis of some specified scheduling discipline. A simulation is often concerned with the relation between scheduling discipline and the acceptability of response times.

In GPSS, queues are not represented explicitly, but there is an implicit queue associated with each FACILITY or STORAGE. In SIMULA 67, a queue is represented by the built-in SIMSET class. In SIMONE, where a resource is represented as a monitor, a queue of processes waiting for the resource can be represented as a condition variable local to the monitor. For example, local to the FOREMAN monitor of section 2.1, there might be declared

```
FREE : INTEGER;
```

which contains the number of free machines, or (if none) the negative of the number of waiting orders, and

```
Q : CONDITION;
```

representing the queue on which the orders wait. Now the procedures of the foreman could be written:

```
PROCEDURE REQUEST;
   BEGIN FREE := FREE-1;
       IF FREE<0 THEN Q.WAIT
   END

PROCEDURE RELEASE;
   BEGIN FREE := FREE+1;
   IF FREE≤0 THEN Q.SIGNAL
   END
```

The command Q.WAIT suspends the process which called REQUEST; the command Q.SIGNAL causes resumption of the process (if any) which earliest executed Q.WAIT. Thus a condition variable Q implements a policy of "first in first out" (fifo) scheduling. Other scheduling disciplines can be specified by a "scheduled condition".

2.5 STATISTICS.

A simulation program is often in principle non-terminating, in the sense that there is no well-defined final state in which it can be said to have arrived at "the answer". Instead, one generally allows a simulation to cycle through a given number of operations; or, alternatively, to execute for a given duration of simulated time. Hence, its state when it terminates is unpredictable; and even if it were, it would be of no real interest. Instead, one is interested in the history of the states through which the simulation has passed. Two runs of the same model ending in the same state but having different histories are not considered as equivalent.

The history of the execution of a simulation is simply the set of "values" of all the components of the simulation at each moment of simulated time. In general, however, the entire set is not of interest and some subset must be selected and summarised to produce the required set of statistics. One method of doing this is to accumulate a histogram of relevant observations. Some special purpose languages would contain such a facility built-in; but in SIMULA 67 it must be programmed in the language itself, using the general-purpose structure provided by the class. For example, suppose that a histogram requires three parameters.

(1) N, the number of intervals.

(2) LOW, the lower bound of the lowest interval; it should be non-negative.

(3) HIGH, the upper bound of the highest interval; it should be greater than LOW.

The class provides two procedures:

(4) RECORD(X), which records the observation X in the histogram; X should be between LOW and HIGH. (LOW ≤ X < HIGH)

(5) PRINT, which prints the histogram in some suitable graphic representation (which we will not specify here).

The entire class can be constructed:

```
CLASS HISTOGRAM (N,LOW,HIGH);
  INTEGER N, LOW,HIGH;

BEGIN INTEGER COUNT;
  INTEGER ARRAY HISTO [0:N-1];

  PROCEDURE RECORD(X); INTEGER X;
     BEGIN INTEGER I;
       I := (X-LOW)÷(HIGH-LOW);
       HISTO[I] := HISTO[I]+1;
       COUNT := COUNT+1
     END;

  PROCEDURE PRINT
     ... body of print ...;

FOR COUNT := 0 STEP 1 UNTIL N-1 DO
       HISTO[COUNT] := 0;
COUNT := 0
END;
```

However, extensive statistics gathering written by the programmer tends to clutter the program and obscure the model. Hence, there is a case to be made for certain semi-automatic facilities (as in GPSS), though this option carries with it the problem that the volume of (perhaps unwanted) statistics can become quite large.

2.6 RANDOM NUMBERS.

In a simulation program, the parameters of a process (e.g. its start time, service times) may be specified in the normal way by the program itself or by its input data; but it is often more convenient to select them at random in accordance with some known or conjectured distribution (e.g. a uniform

distribution between given limits, or a negative exponential with a given mean). A language like GPSS provides a range of random number drawing facilities to assist in the construction of probabilistic models; but in a general purpose language these facilities need not be built-in, since they can be programmed by a pseudorandom multiplicative technique, using some suitable MULTIPLIER and LIMIT. For example, a generator of a random number between zero and one can be implemented as a SIMULA 67 class, and used by repeated calls on the SAMPLE procedure:

```
CLASS RANDOM (SEED); INTEGER SEED;

  BEGIN REAL PROCEDURE SAMPLE;
        BEGIN SAMPLE := SEED/LIMIT;
              SEED := SEED×MULTIPLIER;
              SEED := SEED-(SEED÷LIMIT)×LIMIT;
        END
  END
```

This simple multiplicative algorithm is chosen merely for the sake of the example.

2.7 SUMMARY.

Table 1 shows how the six essential features of discrete event simulation are represented in three languages designed for the purpose. The fourth column provides a comparison with the language described later in this paper. It can be seen that GPSS provides the widest range of built-in special-purpose features, and CSP leaves the most to be programmed in the language itself, possibly with some loss of convenience and efficiency.

This could in fact be done, on condition that:

(1) the TIMER maintains a count of all quasiparallel processes in the system; i.e., all those which make calls of HOLD.

(2) No quasiparallel process communicates with any other process ~~but~~ except the TIMER.

These restrictions are required to ensure that the TIMER can detect when the number of processes which have executed a HOLD is equal to the total number of quasiparallel processes, so that it can advance simulated time and resume the process which is due to be resumed the earliest. However, the restrictions are unacceptably severe; and unless some general-purpose method can be found of triggering the TIMER process when there is nothing else left to do, it would seem necessary to include an automatic timer as a built-in feature of a special-purpose language, intended for discrete event simulation.

One minor extension has been made to the language described previously, in that output commands are permitted to appear as guards in alternative and repetitive commands. This gives a useful increase in the convenience of use of the language, although it may lead to implementation problems on multiple processors with disjoint stores.

3.1 ASSIGNMENT COMMANDS.

(1)	IN := IN+1	adds one to IN
(2)	(N,LOW,HIGH) := (10,0,100)	a multiple assignment, assigning to each target variable on the left, the value of the corresponding element on the right
(3)	REQUEST(n) := REQUEST(3)	the tags "REQUEST" on the left and right are matching, so the effect is the same as n := 3

	GPSS	SIMULA 67	SIMONE	CSP
Resources	FACILITY, STORAGE	ALGOL data structures, CLASS	PASCAL data structures, MONITOR	PASCAL data structures, PROCESS
Processes	TRANSACTION	PROCESS CLASS	PROCESS	PROCESS
Time	ADVANCE	Built-in HOLD	Built-in HOLD	Built-in HOLD
Queues	implicit	SET	CONDITION variable	Programmer responsibility
Statistics	Built-in feature	Programmer responsibility	Programmer responsibility	Programmer responsibility
Random Numbers	Built-in generators	Built-in generators	Built-in generators	Programmer responsibility

Table 1.

3. COMMUNICATING SEQUENTIAL PROCESSES.

A complete description of communicating sequential processes has been given in a previous paper [1]. This section contains a series of annotated examples, selected from the area of discrete event simulation. They may serve as revision for a reader who is already familiar with the previous paper; otherwise, the reader is recommended to study the previous paper, supplementing or replacing its examples by those of this section.

The main difference between the general-purpose language described previously, and the special-purpose language described here is that here the processes are interpreted as being executed in simulated time (in quasiparallel) instead of in real time (genuine concurrency). The mechanism of simulated time was described in 2.3. The question arises whether simulated time could have been implemented as a TIMER process, using only the general-purpose features of the language.

(4) GRANTED( ) := GRANTED( ) — the assignment of matching signals has no effect

(5) ACQUIRE( ) := RELEASE( ) — fails, owing to mismatch of tags

3.2 PARALLEL COMMANDS.

(1) [Q:: queue||U:: user]

Here, "queue" and "user" stand for command lists which are to be executed concurrently, and Q and U are identifiers which name these processes. The processes start simultaneously, and the parallel command ends successfully only if and when both of them have successfully terminated.

(2) [FOREMAN (J:1..10):: foreman]

Here "foreman" stands for a command list, possibly containing the bound variable J. This example specifies ten processes, with names FOREMAN (1), FOREMAN (2), ... FOREMAN (10). The actions of each are specified by the identical text "foreman", except that the value of J in each process gives the index of its name.

3.3 INPUT AND OUTPUT COMMANDS.

(1) READER?STARTTIME - from the READER input an integer value, and assign it to STARTTIME.

(2) PUNCH!"*" - to the punch, output the character "*"

(3) U?(N,LOW,HIGH) - from process named U, input a group of three values, and assign them to variables N, LOW, and HIGH.

(4) HISTOGRAM!(10,0,100) - to process HISTOGRAM, output the three values 10,0,100

Note: if a process named HISTOGRAM issues command (3), and a process named U issues command (4), these are executed simultaneously, and have the same effect as the structured assignment:

(N,LOW,HIGH) := (10,0,100),

i.e. N := 10; LOW := 0; HIGH := 100

(5) ALLOC!ACQUIRE( ) - to process ALLOC send a signal ACQUIRE( )

(6) U?ACQUIRE( ) - from process named U, accept a signal ACQUIRE( )

(7) ORDER(I)?REQUEST( ) - from the i^{th} element of an array of ORDER processes accept a signal REQUEST( )

3.4 ALTERNATIVE AND REPETITIVE COMMANDS.

(1) [FREE<0 → Q!I
[]FREE≥0 → ORDER(I)!GRANTED( )
]

If FREE is negative, I is output to Q; otherwise a GRANTED( ) signal is sent to ORDER(I).

(2) I := 1;
*[I≤NOFSTEPS → ...; I := I+1]

The body ... is repeated NOFSTEPS times, once for each value of I between 1 and NOFSTEPS inclusive.

```
(3)  *[U!(SEED/LIMIT) →
          SEED := (SEED×MULTIPLIER) MOD LIMIT;
      ]
```

Repeatedly outputs a number to U, and then computes a new value of SEED. Terminates when U terminates.

```
(4)  *[U?RELEASE( ) → FREE := FREE+1
      □FREE>0; U?ACQUIRE( ) → FREE := FREE-1
      ]
```

Each repetition either accepts a RELEASE( ) signal from U and then adds one to FREE, or it accepts an ACQUIRE( ) signal from U and then subtracts one from FREE;- but this second alternative can occur only if FREE is originally greater than zero. Thus FREE can never go negative.

```
(5)  *[(I:1..100)ORDER(I)?ACQUIRE( )
                → ORDER(I)?RELEASE( )
      ]
```

Repeatedly accepts ACQUIRE( ) signals from any one of 100 ORDER processes. The bound variable I gives the index of the acquiring ORDER on each occasion. The body of the loop accepts a RELEASE( ) signal from the same I^{th} ORDER. The repetitive command terminates when all hundred ORDER processes have terminated.

```
(6)  *[IN<OUT+100; U?BUFFER(IN MOD 100) →
                    IN := IN+1
      □OUT<IN; U!BUFFER(OUT MOD 100) →
                    OUT := OUT+1
      ]
```

Repeatedly, on request from U,

either (1) (Provided that IN<OUT+100) inputs a value from U, and stores it in the appropriate element of an array BUFFER

or (2) (Provided that OUT<IN) outputs the value of the appropriate element of BUFFER to U.

The repetitive command terminates when U does.

4. EXAMPLES.

In this section, we present a series of examples to show how communicating sequential processes can be used to implement the basic requirements of discrete event simulation, as described in section 2; but the topics are treated in the opposite order.

4.1 RANDOM NUMBERS.

Problem: Write a process RANDOM to represent a stream of random numbers, as described in section 2.6. The name of the using process is U. The process first inputs from U the value of its seed, and then it outputs a series of random numbers starting with one derived directly from the seed.

Solution:

```
RANDOM::
[SEED: INTEGER; U?SEED; SEED := SEED MOD LIMIT;
 *[U!(SEED/LIMIT) →
        SEED := (SEED×MULTIPLIER) MOD LIMIT
] ]
```

4.2 STATISTICS: HISTOGRAM.

Problem: Write a process to represent a histogram, as described in section 2.5. The name of the using process is U. The histogram first inputs its parameters N, LOW, and HIGH; it then inputs and records a series of integers from U. When U terminates, the histogram is automatically printed in some suitable graphic notation. If any input value is invalid, the process aborts.

Solution:

```
HISTOGRAM::
[N,LOW,HIGH, COUNT: INTEGER;
 U?(N,LOW,HIGH); COUNT := 0; [N>0 & 0 ≤LOW& LOW < HIGH →SKIP];
 HISTO: ARRAY(0..N-1) OF INTEGER;
 FOR I = 0..N-1 DO HISTO(I) := 0; X: INTEGER;
*[~~X:INTEGER;~~ U?X → I:INTEGER; [LOW≤ X& X<HIGH →SKIP];
        I :=⌊(X-LOW)÷(HIGH-LOW);
        HISTO(I) := HISTO(I)+1;
        COUNT := COUNT+1
 ]; ... print the value of HISTO ...
]
```

N×⌊

4.3 QUEUES: A FIFO DISCIPLINE.

Problem: Write a process Q to implement a fifo queue of integers for a user U. The user appends an integer I to the queue by an output command Q!I. It removes the first member of the queue by the input command Q?F which assigns to F the value removed; this will be delayed ~~fail~~ if the queue is empty. The maximum length of the queue is 100.

Solution:

```
Q:: IN,OUT: INTEGER; IN := 0; OUT := 0;
BUFFER: ARRAY (0..99) OF INTEGER;
*[IN<OUT+100; U?BUFFER(IN MOD 100) →
                    IN := IN+1
 []IN>OUT; U!BUFFER(OUT MOD 100) →
                    OUT := OUT+1
 ]
```

4.4 SINGLE RESOURCE ALLOCATOR.

Problem: A single input device is to be shared among an array of processes

```
ORDER(I: 1..100):: ...
```

Each order acquires the device by a command

```
ALLOC!ACQUIRE( );
```

it then uses the device, and finally releases it by:

```
ALLOC!RELEASE( );
```

Write the process ALLOC, which ensures that at most one ORDER at a time can use the device.

Solution:

```
ALLOC::
*[(I:1..100) ORDER(I)?ACQUIRE( ) →
                ORDER(I)?RELEASE( )
 ]
```

4.5 MULTIPLE RESOURCE ALLOCATOR.

Problem: Write a process FOREMAN to allocate 10 machines among an array of 100 processes:

ORDER(I: 1..100):: ...

An order acquires a machine by a pair of commands:

FOREMAN!REQUEST( ); FOREMAN?GRANTED( );

and it releases a machine by

FOREMAN!RELEASE( )

When there are no free machines, the foreman uses a fifo queue to store the identity of the orders whose requests cannot yet be granted.

Solution:

```
FOREMAN::
[Q:: see example (4.3) ...
||U:: FREE: INTEGER; FREE := 10;
  *[(I: 1..100) ORDER(I)?REQUEST( ) →
          FREE := FREE-1;
          [FREE<0 → Q!I
          ▯FREE≥0 → ORDER(I)!GRANTED( )
          ]
   ▯(I: 1..100) ORDER(I)?RELEASE( ) →
          FREE := FREE+1;
          [FREE>0 → SKIP
          ▯FREE≤0 → F:INTEGER;
                    Q?F; ORDER(F)!GRANTED( )
          ]
  ]
]
```

4.6 MACHINE SHOP. [3].

A machine shop contains ten groups of ten machines each. Each group of machines is scheduled by a foreman using a fifo discipline. The machine shop must process a hundred orders. Each order has the following parameters:

1. STARTTIME - the simulated time at which the order enters the shop for processing.
2. NOFSTEPS - the number of steps required for processing the order.
3. For each step, numbered between 1 and NOFSTEPS, there are two parameters:

   (3.1) MACHGROUP - the number of the machine group required to carry out this step

   (3.2) SERVICETIME - the amount of time required to process this step.

   These parameters for each order may be read from the input device READER. Each order must acquire exclusive access to the reader before ~~doing so~~ reading its parameters.

Solution: The overall structure of the solution is:

```
[ALLOC:: ... see example(4.4)...
||U:: [(J: 1..10) FOREMAN:: ... see example(4.5)...
    ||(K: 1..100) ORDER:: ... see below ...
     ]
]
```

The order process array is:

```
   ORDER(K: 1..100)::
comment read the parameters for this order;
   ALLOC!ACQUIRE( ); STARTTIME,NOFSTEPS: INTEGER;
   READER?STARTTIME; READER?NOFSTEPS;
   MACHGROUP,SERVICETIME: ARRAY(1..NOFSTEPS) OF INTEGER;
   I: INTEGER; I := 1;
   *[I≤NOFSTEPS → READER?MACHGROUP(I);
                  READER?SERVICETIME(I);
                  I := I+1
    ]; ALLOC!RELEASE( );
comment start the simulation proper;
   HOLD(STARTTIME); I := 1;
   *[I≤NOFSTEPS → J: INTEGER; J := MACHGROUP(I);
                  FOREMAN(J)!REQUEST( );
                  FOREMAN(J)?GRANTED( );
                  HOLD(SERVICETIME(I));
                  FOREMAN(J)!RELEASE( );
                  I := I+1
    ];
```

5. CONCLUSION.

This paper has shown by example that the general purpose concept of a communicating sequential process is adequate for many of the requirements of a special purpose discrete event simulation language, provided that the concept of simulated (quasiparallel) time is also built into the language. Whether this too can be implemented by some reasonable general-purpose feature is an open question.

The notations described and used in this paper are not recommended for general use as a programming language, since they still suffer from many of the defects summarised in [1], namely,

(1) The static upper bound on the size of an array, including an array of processes. This defect has been masked in the example problems by artificial simplification.

(2) The absence of aids to the construction and use of libraries of standard processes.

(3) The non-existence of an efficient implementation.

Suggestions for the solution of these problems have not been given in this paper.