SIMULA—an ALGOL-Based Simulation Language

OLE-JOHAN DAHL AND KRISTEN NYGAARD
Norwegian Computing Center, Oslo, Norway

This paper is an introduction to SIMULA, a programming language designed to provide a systems analyst with unified concepts which facilitate the concise description of discrete event systems. A system description also serves as a source language simulation program. SIMULA is an extension of ALGOL 60 in which the most important new concept is that of quasi-parallel processing.

1. Introduction

SIMULA (SIMUlation LAnguage) is a language designed to facilitate formal description of the layout and rules of operation of systems with discrete events (changes of state). The language is a true extension of ALGOL 60 [1], i.e., it contains ALGOL 60 as a subset. As a programming language, apart from simulation, SIMULA has extensive list processing facilities and introduces an extended co-routine concept in a high-level language.

The principal features of SIMULA are defined below. The syntax rules given here are slightly simplified versions of the actual rules, since our intention is merely to introduce what we feel are the most important ideas of the language. For a complete definition of SIMULA, see [2].

A SIMULA compiler has been in operation on the UNIVAC 1107 computer since January 1965. The compiler translates a SIMULA system description into an object code simulation program for the system described. The compiler has now been used for the analysis of a great number of problems. Compiling and execution speeds are comparable to those of ALGOL and FORTRAN programs of corresponding complexity.

SIMULA has been designed and implemented by the authors at the Norwegian Computing Center under a contract with the Univac Division of Sperry Rand Corporation.

1.1 Design Objectives

Simulation is now a widely used tool for analysis of a variety of phenomena: nerve networks, communication systems, traffic flow, production systems, administrative systems, social systems, etc. Because of the necessary list processing, complex data structures and program sequencing demands, simulation programs are comparatively difficult to write in machine language or in ALGOL or FORTRAN. This alone calls for the introduction of simulation languages.

However, still more important is the need for a set of basic concepts in terms of which it is possible to approach, understand and describe all the apparently very different phenomena listed above. A simulation language should be built around such a set of basic concepts and allow a formal description which may generate a computer program. The language should point out similarities and differences between systems and force the research worker to consider all relevant aspects of the systems. System descriptions should be easy to read and print and hence useful for communication.

The need for the inclusion of algorithmic procedures as parts of a discrete event system description makes it natural to let a simulation language contain an algorithmic language as a subset. Efforts then should be made, not only to ensure that the simulation language is a logical extension of the algorithmic language, but also to achieve an increase in the power of the algorithmic language as such.

1.2 Basic Concepts

An ALGOL program (block) specifies a sequence of operations on data local to the program, as well as the structure of the data themselves. SIMULA extends ALGOL to include the notion of a collection of such programs, called "processes," conceptually operating in parallel. The processes perform their operations in groups called "active phases" or "events." Between any two consecutive active phases of one process any number of active phases of other processes may occur. The sequence of operations in the system

The paper was presented as the basis of a lecture given at a Seminar on Simulation Languages at Centro Nazional Universitario di Calcolo Elettronico, Pisa, Italy, May, 1966. An earlier, less comprehensive version was presented at a NATO "Conference on the Role of Simulation in Operations Research," Hamburg, Germany, September, 1965.

as a whole thus becomes a sequence of active phases of the processes present in the system. This mode of operation will be called "quasi-parallel."

The following are the main extensions to ALGOL, which are introduced:

(1) Means of describing processes, generating processes dynamically, and referencing existing ones.
(2) Means of delimiting and sequencing active phases of processes, with or without reference to the concept of "system time."
(3) Means of making data local to processes accessible from other processes.
(4) An ordered set mechanism convenient for queueing and stacking processes.

The SIMULA concepts are introduced partly by extensions to the ALGOL 60 syntax, partly through system procedures. Among the latter are a number of procedures for random drawing from mathematically or empirically defined distribution functions and for accumulating system time integrals and histograms.

2. Processes

The process concept is intended as an aid for decomposing a discrete event system into components, which are separately describable. In general, a process has two aspects: it is a data carrier and it will execute actions. The items of data carried by a process are described by a sequence of specifications and declarations; the actions are described by a sequence of statements called its operation rule. The two aspects are combined into a block called a *process block.*

The description of a process is called an *activity* declaration. The concept of an "activity," which is a class of processes described by the same declaration, is distinguished from the concept of a "process," which is one dynamic instance of an activity declaration. The declaration is syntactically similar to that of a procedure.

⟨activity identifier⟩ ::= ⟨identifier⟩
⟨activity body⟩ ::= ⟨statement⟩
⟨activity declaration⟩ ::= **activity** ⟨activity identifier⟩
⟨formal parameter part⟩; ⟨specification part⟩ ⟨activity body⟩

A process block is the activity body itself, if the latter takes the form of an ⟨unlabelled block⟩; otherwise it is an implied block enclosing the activity body. The parameters and the items declared local to the process block are called the *attributes* of a process; its operation rule is the list of statements within the process block.

Unlike procedures, which are dynamically nested, the relationship between processes is a symmetric one. They operate in a quasi-parallel fashion, as defined below.

A discrete event system will be viewed as a collection of processes, whose actions and interactions comprise the behavior of the system. Processes will enter and leave the system as results of actions within the system itself.

2.1 PROCESS REFERENCES

Processes can be referenced individually. Physically, a process reference is a pointer to an area of memory containing the data local to the process and some additional information defining its current state of execution. However, for reasons stated in the Section 2.2 process references are always indirect, through items called "elements." Formally a reference to a process is the value of an expression of type **element.**

⟨element expression⟩ ::= **none** | ⟨variable⟩ | ⟨function designator⟩ |
⟨process designator⟩ | ⟨activity identifier⟩
⟨process designator⟩ ::= **new** ⟨activity identifier⟩
⟨actual parameter part⟩

element values can be stored and retrieved by assignments and references to **element** variables and by other means.

A process is generated by evaluating a *process designator.* Its value is a reference to the generated process. A process will remain part of the system *as long as it can be referenced* through a computable **element** expression. (The deletion is automatic and is effected through a combination of the "reference count" and "garbage collection" techniques; see [6].)

Example.

```
element Pat;
activity secretary (redhaired, thumbs);
        Boolean redhaired; integer thumbs;
begin - - - end;
Pat := new secretary (true, 10);
```

The value of the variable Pat is now a reference to the generated secretary process. If another value is later assigned to Pat, and if there is no other way of referencing this secretary, she is deleted.

The symbol **new** serves to distinguish between different uses of activity identifiers. As described in Section 4 an ⟨activity identifier⟩ may be used as an **element** expression, without the symbol **new**, within a "connection block."

The language contains a mechanism for making the attributes of a process accessible from the outside, i.e., from within other processes. This is called remote accessing. A process is thus a *referenceable data structure.*

It is worth noticing the similarity between a process whose activity body is a dummy statement, and the record concept recently proposed by C. A. R. Hoare and N. Wirth [3]. Since processes can reference one another through local **element** variables, they have the same list forming capabilities. The ordered set mechanism described in the next section has been added for convenience. The mechanism for "remote accessing" in SIMULA differs from the one proposed in [3].

2.2 ELEMENTS AND SETS

The element and set concepts[1] serve to facilitate and standardize the manipulation of queues and other linear

[1] Many of the ideas presented in this section were inspired by the SLIP system [5].

lists of processes. A set is an ordered sequence of *elements*, which are objects having a standard format as shown in Figure 1. An element pointing to a process is a marker representing that process. There may be more than one

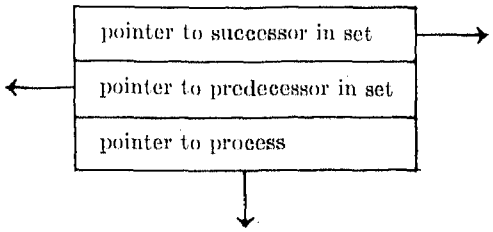

FIG. 1. Element in a set. In addition to the information shown, an element contains a "reference count," which implicitly defines its "lifetime" in the system.

element in the system pointing to the same process. A process is said to be member of a set if there is an element in the set pointing to it.

A value of type **element**, except the value **none**, is a pointer to an element. We may equivalently say that the value is the element itself, and we shall do so in order to shorten the following discussion.

Two of the three components of an element (Figure 1) are themselves **element** values. They are accessed through basic **element** procedures. If X is an element in a set, *suc*(X) is the successor element and *pred*(X) is the predecessor. These pointers between elements of a set are updated implicitly by system procedures such as *include*(X) *in:*(S), *precede*(X)*by:*(Y), and *remove*(X), where X and Y are elements and S is a set. If an element X currently has no set membership, the values of *suc*(X) and *pred*(X) are both **none**.

The third component of an element, the process pointer, remains fixed. It is not itself an **element** value, and is therefore not explicitly accessible as the value of an expression. There is, however, a corresponding **element** procedure "*proc*," which generates an element. The value of *proc*(X) is a new element with the same process pointer as X and no set membership. As shown above, a ⟨process designator⟩ is another "generating" expression, which actually generates both a process and an element referring to it. The latter has no set membership and is the value of the expression.

The process pointer of an element is used implicitly by a number of system procedures and special statements, which operate on the process referenced by a specified element. The element concept and the technique of referencing processes indirectly yield the following desirable language properties.

(1) Ordered sets of processes can be manipulated by means of efficient standard procedures.

(2) When a process is referenced through an element in a set, its successor and predecessor in the set are immediately accessible.

(3) Any given process can be member of an unlimited number of sets at the same time.

(4) The members of a set can be processes of different classes.

A set has one permanent dummy element called the *set head*, which conforms to the standard element format, except that it refers to no process. Together with the ordinary elements of a set it forms a circular list of elements (Figure 2). An "empty" set has a set head which is its own successor and predecessor.

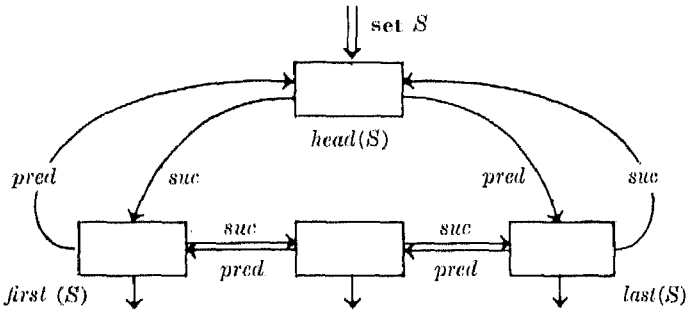

FIG. 2. Set with 3 members

A set designator

⟨set designator⟩ ::= ⟨variable⟩

has a value of type **set**, which is, physically, a pointer to a set head. This pointer remains fixed during the scope of the set designator. The set head is generated as the result of the set declaration, at which time the set is empty. At the end of the scope of a set each element loses its set membership. The head of a set S can be referenced as an element through the function designator "*head*(S)." The "first" and "last" elements of the set are defined as

$$first(S) = suc(head(S))$$

$$last(S) = pred(head(S)).$$

The members of a set S may be scanned successively by means of the following **for** clause:

for X := *first*(S), *suc*(X) **while** *exist*(X) **do**

or, in order to avoid the execution of the controlled statement if the set is empty:

X := *head*(S); **for** X := *suc*(X) **while** *exist*(X) **do**

The Boolean expression "*exist*(X)" has the value **true** if X refers to a process, otherwise **false**. The set head is the only element of a set which does not refer to a process thereby signalling the "end" of the set in either direction.

2.3 The Sequence Control

The actions of a process are grouped together in active phases, separated by periods of inactivity. Only one process is active executing actions at any one time. An inactive period of a process is caused by a deactivating statement executed by that process. (Deactivating statements are special cases of the sequencing statements described in Section 2.6.) The statement terminates the current active phase, and control leaves the process. Usually a *reactivation point* is defined for the process, whose function is similar to that of the "router" of a co-routine, as defined by M. E. Conway [4]. At the time of the next active phase of the process, control re-enters the process at the place where it most recently left, as specified by the reactivation point. (A deactivating statement may be given inside any sub-block or procedure called by the process, therefore the reactivation point must also include information relating to the status of a "block stack" belonging to the process, as used in most Algol implementations.)

The reactivation point concept allows the programmer to string together actions occurring at widely different times into a logically coherent sequence. He can think of a process as possessing its own *"local" sequence control*, identical to the main control during active phases, and represented by the reactivation point during inactive periods. From this local point of view a deactivating statement allows other processes to be active during its execution.

An active phase of a process is called an *event*. It has an associated *system time*, which remains constant during the execution of the event. The system times form a non-decreasing sequence of **real** numbers. A deactivating statement may invoke an inactive period of a definite or indefinite length. As an example of the former, consider the following activity declaration.

```
activity report(dt); real dt;
begin L: write (-----); hold(dt); go to L end;
```

The statement *"hold(dt)"* is a call to a system procedure, which represents an inactive period of length *dt* in system time. It follows that a report process will give output with regular system time intervals. Several report processes could operate in quasi-parallel, e.g., **new** *report*(7) and **new** *report*(30) giving "weekly" and "monthly" reports, provided that the system time is being given in units of one day.

2.4 The Sequencing Set (SQS)

An event can be scheduled to happen either immediately or at some later time. The event is the next active phase of some specified process. A process for which an event has been scheduled but not completed has an associated *event notice* representing the event. There can be at most one event notice associated with a given process.

An event notice contains a reference to the associated process (through an element) and a **real** number, called its time reference. The event notices are members of the *sequencing set* (SQS), which is ordered according to non-decreasing time references, as shown in Figure 3. The time reference of an event notice is accessed through the **real** function *evtime*(*X*), where *X* is an **element** expression referencing the associated process. The sequencing set is not, strictly speaking, a "set" in the sense of Section 2.2.

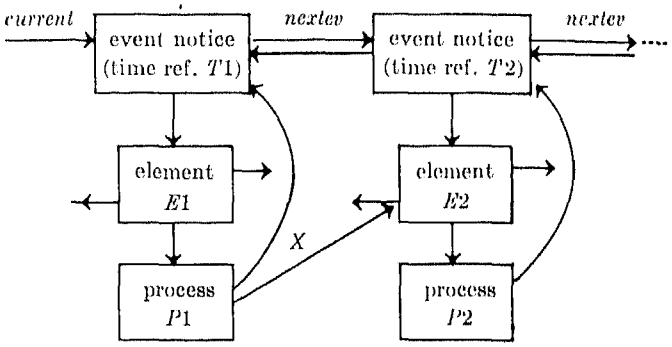

Fig. 3. The sequencing set. *P*1 is the currently active process, *P*2 is suspended, "current" is equal to *E*1 and "time" is equal to *T*1. *P*1 has a local **element** variable *X*, to which has been assigned the value "nextev(current)". "evtime(X)" is equal to *T*2, which is greater than or equal to *T*1.

The event notice at the "lower end" of the SQS is called the *current* event notice. Its associated process is the currently active one, and its time reference is regarded as the current value of the "system time." The active process may be referenced by the **element** procedure *"current,"* and there is a **real** procedure *"time"* equivalent to *"evtime(current)."*

When the current active phase is completed, the current event notice is deleted. Its successor in the SQS becomes the current event notice, and control enters the associated process at its reactivation point.

2.5 States

A process can be in one of four possible *states*: active, suspended, passive, and terminated. As simulation proceeds, the states of processes will change.

(1) *Active.* The currently active process can, by *sequencing statements*, alter the states of processes, including its own.

(2) *Suspended.* A suspended process has an associated event notice and a reactivation point. Unless a change of state is caused by another process, the next active phase of this process will start when the event notice becomes the current one.

(3) *Passive.* A passive process has a reactivation point, but no associated event notice. It will remain passive until a change of state is caused by another process. At

the time of generation a process is passive, and its reactivation point is in front of the first statement of its operation rule.

(4) *Terminated.* A process becomes terminated whenever control passes through the final **end** of its operation rule or exit from the process is made by a **go to** statement. Such a process has no reactivation point and no event notice. The state of a terminated process can not be altered by ordinary sequencing statements.

A process will remain part of the system at least as long as it has an associated event notice. It can be referenced through system procedures such as "*current*" if active, or "*nextev*(X)" if suspended (where X refers to an active or suspended process). A passive or terminated one will remain (as a data structure) as long as it can be referenced through an **element** expression.

2.6 Sequencing Statements

Sequencing statements are statements operating on the SQS, thereby altering the states of processes. A sequencing statement may delete an event notice and/or schedule an event by generating an event notice and include it in the SQS.

The basic sequencing statements are:

(1) *cancel* (⟨element expression⟩), which deletes the event notice, if any, associated with the referenced process,

(2) *terminate* (⟨element expression⟩), which in addition deletes the reactivation point, if any,

(3) scheduling statements, which are described below.

Deletion of the current event notice causes the current active phase to terminate. Any sequencing statement, except "terminate" will in that case define a reactivation point for the process at the end of the statement. The statement "*terminate(current)*" is equivalent to leaving the process through its final **end.** The statement "*cancel (current)*" represents an inactive period of indefinite length. The process becomes passive.

A scheduling statement usually generates an event notice for a specified process and includes it in the SQS. The process is referenced through an element expression, whose value is stored in the event notice.

In addition, the statement will specify explicitly either (1) the time reference of the event notice, or (2) its position in the SQS. The information not specified is determined according to rules consistent with the ordering criterion of the SQS.

The basic scheduling statements are special syntactic constructions.

⟨activator⟩ ::= **activate** | **reactivate**
⟨simple timing clause⟩ ::= **at** ⟨arithmetic expression⟩ |
delay ⟨arithmetic expression⟩
⟨timing clause⟩ ::= ⟨simple timing clause⟩ |
⟨simple timing clause⟩ **prior**
⟨scheduling clause⟩ ::= ⟨empty⟩ | ⟨timing clause⟩ |
before ⟨element expression⟩ | **after** ⟨element expression⟩
⟨scheduling statement⟩ ::= ⟨activator⟩ ⟨element expression⟩
⟨scheduling clause⟩

The activator **activate** will cause the generation of an event notice only if the referenced process is passive, whereas **reactivate** in addition will delete the event notice associated with an active or suspended process and "re-schedule" the event.

A timing clause specifies the time reference of the generated event notice, either "*time*" or the value of the expression, whichever is greater, and this determines its position in the SQS. The event notice is normally placed behind all others with the same time reference, but it can also be placed in front of these event notices by adding the symbol **prior.**

The statement "*hold*(T)" already mentioned is equivalent to:

reactivate *current* **delay** T (or "**at** *time* + T")

Another option is to specify the position of the event notice relative to (**before,** or **after**) the event notice associated with a second process. It gets the same time reference as the latter. If the second process is passive or terminated, no scheduling takes place.

The scheduling clauses "**delay** 0 **prior**", "**at** *time* **prior**", "**at** 0 **prior**" and "**before** *current*" are equivalent. The generated event notice is placed in front of the current event notice, whereby the currently active process becomes suspended. The scheduled event becomes the current event. This is called "direct scheduling". Since direct scheduling is used frequently in practice, an empty (omitted) scheduling clause is understood to be an abbreviation for direct scheduling.

The statement "**activate** ⟨element expression⟩" is therefore somewhat similar to a procedure call, in the sense that actions are invoked as a "subroutine" to the calling program. Notice, however, that the event notice associated with the calling process may well be deleted or rescheduled before control returns.

Two processes X and Y, calling each other by, respectively, **reactivate** Y and **reactivate** X, would function as co-routines in the sense of [4]. They could communicate through nonlocal variables.

2.7 Parameters and Nonlocals

The quasi-parallel operation of processes is not compatible with the simple stack structure of an Algol program at run time. In order to make possible an efficient data storage allocation scheme the activity declaration is subject to certain restrictions as compared to the procedure declaration.

(1) Only ⟨type⟩ and ⟨type⟩ **array** parameters are permitted. The former are transmitted by value, the latter by location. The activity declaration has no ⟨value part⟩.

(2) Activity declarations are only accepted in the block heads of certain special blocks called Simula blocks; this is to permit unrestricted reference to nonlocals within processes.

2.8 Main Program

A Simula block

⟨Simula block⟩ ::= **SIMULA** ⟨unlabelled block⟩

is itself a statement, which can be part of an otherwise ordinary Algol program. Simula concepts are not available outside a Simula block. The SQS is local to it and all activity declarations describing a discrete event system are given in the Simula block head.

The Simula block functions dynamically as a process called the "main program," and at the same time as an outer block to all processes except itself. It is always present in the system. Upon entry into a Simula block the main program is the only process in the system. It is active, and system time is zero. When exit is made out of the Simula block the simulation is terminated.

The initialization of the system is usually done by the main program. It may suspend itself during the simulation proper, or may take an active part in it.

3. An Example

The following program is a skeleton Simula description of a classical "job shop" system. The shop consists of a given number (*nmg*) of machine groups. The *j*th group contains a number of identical machines, given by the initial value of "*available* [*j*]." They operate in parallel and have a common queue (*que*[*j*]) of orders waiting. The orders are treated on a "first come first served" basis at each group.

An order defines its own path through the shop (*mgroup*[1], ···, *mgroup*[*n*]) and its processing time at each group of the path (*ptime*[1], ···, *ptime*[*n*]). The arrival pattern of orders and their contents are defined by an input stream. The data describing an order are the number n of steps in its schedule, its arrival time T, and the schedule itself represented by the components of the arrays *mgroup* and *ptime*. The end of the arrival pattern is signalled by a zero n, where the associated T is the system time at which the simulation is to be terminated.

The system might be formulated as a collection of machine processes acting on orders (through remote accessing). However, in this case, the simplest program is obtained by looking at the system entirely from the point of view of an order, since the latter contains all the information relevant to the routing and processing. The main program acts as a steering routine timing the order arrivals.

The following program contains only statements essential to the behavior of the system. No data are collected, and no output is given.

```
Line
 1. begin integer nmg; read(nmg);
 2. SIMULA begin integer array available [1:nmg];
      set array que [1:nmg];
 3. activity order(n); integer n;
 4. begin integer i, mg; integer array mgroup[1:n];
      array ptime[1:n];
 5.   read(mgroup, ptime);
 6.   for i:= 1 step 1 until n do
 7.   begin mg := mgroup [i];
 8.   if available[mg] = 0 then
 9.   begin wait(que[mg]); remove(current) end
10.   else available[mg] := available[mg] −1;
11.   hold(ptime[i]);
12.   if empty(que[mg]) then available[mg] := available[mg] +1
13.   else activate first (que[mg])
14.   end path through shop
15.   end order;
16. integer n; real T;
17. read(available);
18. next: read(n, T); reactivate current at T;
19. if n > 0 then begin activate new order(n); go to next end
20. end SIMULA end program
```

Comments

Line 2. All sets are empty initially. The machines are all available (cf. line 17).

Line 3. The activity declaration (lines 3–15) describes the structure and behavior pattern of an order.

Line 5. An order reads its own schedule during its first active phase.

Line 6. The controlled variable counts the number of steps in the schedule. The following compound statement (lines 7–14) describes the behavior of an order at each machine group.

Line 9. If there is no available machine in the current machine group (*mg*), the order includes itself in the appropriate queue and goes passive. The system procedure *wait*(*S*) is equivalent to

include(*current*, *S*); *cancel*(*current*).

The order becomes "*last*(*que*[*mg*])". The local sequence control stays at this statement until the order has become "*first*(*que*[*mg*])", and is activated by another order leaving the machine group (line 13). The next statement removes the order from *que*[*mg*].

Line 10. The order allocates an available machine to itself.

Line 11. The hold statement represents the processing time.

Line 12. If the queue is now empty, the machine is made available.

Line 13. If not, this order activates the one in front of the queue. The latter removes itself from the queue (line 9) and becomes suspended (line 11). Control then returns to the former order, which proceeds to the next machine group or leaves the shop.

Line 15. At this point the order is a member of no set, and the only reference to it is from the current event notice. At the end of the active phase the latter is deleted and the order therefore leaves the system.

Line 17. This is the first statement of the Main Program, and therefore it is the first statement of the first event, taking place at system time zero. The sizes of the machine groups are read.

Line 18. The arrival of the next order is timed according to the value read for T. The Main Program suspends itself until system time T.

Line 19. The **element** expression "**new** order(*n*)" generates an order. Its first active phase is scheduled directly, whereby the n-step schedule is read (line 5). At the end of this active phase (line 9 or 11) control returns to the Main Program. If n is nonpositive, the simulation is terminated (at system time T).

4. Remote Accessing

Remote access to items local to a process is made possible through the *connection* mechanism. A connection

statement may have the form:

inspect ⟨element expression⟩ **when** A_1 **do** S_1
when A_2 **do** S_2
⋮
when A_n **do** S_n
otherwise S

where A_i are activity identifiers and S_i and S are statements. If the referenced process belongs to the activity A_i, S_i is executed and the other statements are skipped. The **otherwise** clause is optional. The symbol "**inspect**" may be replaced by "**extract**", which will remove the referenced element from its set, if it has set membership.

S_i acts as if it is immediately enclosed in a block containing exactly the same local items as the process block of the referenced process. Thereby the attributes of the process are accessible within S_i through their local names. S_i is called a *connection block*, and the referenced element and process are said to be *connected*. Within a connection block the relevant activity identifier functions as an **element** procedure, whose value is the connected element.

A connection clause (**when** A **do** S), or otherwise clause (**otherwise** S) within the lexicographic scope of more than one connection statement is part of the one with the smallest scope. An otherwise clause terminates the scope of the corresponding connection statement. (Evidently the scope of a connection statement can also be terminated by any symbol "**end**", "**else**" or ";" which is not a part of the statement.)

The attributes of a process may include procedures, switches and labels. By going to a label local to a connected process it is possible to override its reactivation point and to "revive" a terminated one. [This situation is explained further in 2.]

The connection mechanism has been designed with the following objectives in mind:

(1) To make the language "self-protecting" in the sense that only meaningful references to attributes can be made in a program acceptable to the compiler. The user is forced to check the class of the referenced process.

(2) To give easy and rapid access to the attributes once this check has been made. Interaction between processes often requires remote access to several attributes of the same process, occasionally interspersed with scheduling statements.

(3) To preserve the locality of attribute identifiers. The same identifier can be used for different purposes within different activities. The semantic contents of an attribute identifier within a connection block are defined by the activity identifier of the connection clause.

The next version of the language, now being implemented, will be extended by "**external**" declarations (and specifications) for remote accessing. An **external** declaration declares a nonlocal name for a local quantity (in contrast to **own**, which declares a local name for a nonlocal quantity). Remote access to a quantity local to a given process and declared **external** will be written "⟨element expression⟩.⟨identifier⟩." **external** identifiers will belong to the SIMULA block, and must be unique. It follows that the semantic contents of such an identifier is known to the compiler anywhere within the SIMULA block, and that a simple run-time consistency check can be devised for the process reference.

5. Second Example

The program below is a description of a simple epidemic model. A contagious, nonlethal disease is spreading through a *population* of a given fixed size. (Words used as identifiers in the program are italicized here.) Certain countermeasures are taken by a public health organization.

An individual infection has a given *incubation* period, during which the subject is noncontagious and has no *symptoms*, followed by a contagious period of a given *length*. Each *day* of the latter period the subject may seek *treatment* and get cured. The (unconditional) probability of doing so is *probtreat*[*day*] (*day* = 1, 2, ⋯, *length*). Each sick person has an expected number of *contacts* per day. At one such contact the probability of infecting an uninfected person is *prinf*. A person previously cured is immune. If untreated the infection ceases spontaneously after the given period.

Only *sick persons* appear as processes in the system. When cured they leave the system. The very first infection is generated by the Main Program. A subject infected by another person is included as an element of a set belonging to the latter, called his *environment*. Any subject is a member of at most one such set. It will be seen that the subjects initially form a simple tree structure. As subjects are cured they are removed from the tree, which will therefore disintegrate into smaller tree structures. The latter will grow independently, disintegrate further, and so forth. As the number of *uninfected* persons decreases the growth of the contagion slows down, until finally it dies out.

The public countermeasures are represented by *treatment* processes. A patient is removed from the environment set to which he belongs, if any. If he has visible symptoms, he is cured (terminated). In addition his environment is searched and each member is subjected to a full treatment, which may cause another environment to be searched, etc. A patient displaying no symptoms is given "mass treatment" (cheap pill) which has a certain probability, *probmass*, of success. His environment is not searched. In the present model, treatments act instantaneously in system time. The simulation ends after *simperiod* units of time.

The following program shows only the system description (SIMULA block) itself, and no output is given. The following quantities are declared and given values in outer blocks:

integer *population, length, contacts, u1, u2, u3, u4*;
real *incubation, prinf, probmass, simperiod*;
array *probtreat*[1: *length*];

The variables *u1*, *u2*, *u3*, *u4* represent different "streams" of pseudo-random numbers.

```
1.  SIMULA begin integer uninfected;
2.    activity sick person;
3.    begin integer day; Boolean symtoms; set environment;
4.      uninfected := uninfected-1; symptoms := false;
5.      hold(incubation); symptoms := true;
6.      for day := 1 step 1 until length do
7.      begin if draw(probtreat[day], u1)
          then activate new treatment (current);
8.        infect(Poisson(contacts, u2), environment);
9.        hold(1) end
10.     end sick person;
11.   procedure infect(n, S); value n; integer n; set S;
12.   begin integer i;
13.     for i := 1 step 1 until n do
14.     if draw(prinf×uninfected/population, u3) then
15.     begin include(new sick person, S);
          activate last(S) end
16.   end infect;
17.   activity treatment(patient); element patient;
18.   begin element X;
19.     extract patient when sick person do
20.     if symptoms then
21.     begin terminate(patient);
22.       for X := first(environment) while exist(X) do
23.       activate new treatment(X) end
24.     else if draw(probmass, u4) then terminate (patient)
25.   end treatment;
26.   uninfected := population;
27.   activate new sick person; hold(simperiod)
28. end SIMULA
```

Comments

Line 4. A new sick person has entered the system: the number of persons that have not been in touch with the disease, must therefore be reduced by one.

Line 5. He executes no action relevant to the disease during the incubation period.

Line 6. The controlled statement (lines 7–9) describes his behavior pattern each day of the contagious phase.

Line 7. *"draw"* is a **Boolean** system procedure, which makes a random drawing. The probability of getting **true** as the result is given by the first parameter. If the activate statement is executed the sick person is terminated immediately (line 21).

Line 8. Through the *"infect"* procedure (lines 11–16) he tries to infect a number of other persons, given as a random drawing from the Poisson distribution with mean *"contacts."* *"Poisson"* is an **integer** valued system procedure.

Line 9. Further actions are postponed till the next day. System time is measured in days in this example.

Line 10. If the sick person is never treated, the local sequence control will finally leave the operation rule through **end**, whereby he becomes terminated (cured). He also leaves the system.

Line 14. When a person chosen at random is contacted, the probability of infecting him is *"prinf"* multiplied by the probability that he is previously uninfected. If the value of *draw* is **false** nothing is done, and therefore the identity of the contacted person is of no concern.

Line 15. A sick person is generated and included in *S*, which is the *"environment"* set of the person calling the procedure (line 8). The former becomes *last*(*S*). His first active phase is scheduled directly, whereby the number of uninfected persons is updated (line 4) before the next contact is made.

Line 17. The **element** parameter refers to a sick person process.

Line 19. The patient is removed from the set (*environment*) of which it is a member (if any) and, since it is a *"sick person"* process, it is *connected*, and the following statement is executed. This statement (lines 20–24) is a *connection block*, in which the attributes of the connected sick person are accessible.

Line 20. *"symptoms"* is an attribute of the connected sick person (cf. line 3).

Line 21. The sick person process is terminated; i.e., the event notice and reactivation point belonging to it are both deleted.

Line 22. *"environment"* is an attribute of the connected sick person. The **for** statement amounts to a scanning of this set, because the controlled statement (line 23) removes the current element *X* from the set (line 19 of the new *treatment* process).

Line 24. A patient without symptoms is given mass treatment with probability *"probmass"* of success.

Line 25. The treatment process leaves the system, and so does the sick person, if he has become terminated.

Line 26. The Main Program starts here. Initially the entire population is uninfected.

Line 27. The first infection is generated, after which the Main Program suspends itself for the simulation period.

Acknowledgments. The SIMULA compiler presently operating has been implemented by the authors as an extension of the UNIVAC 1107 ALGOL compiler, under a contract with the UNIVAC Division of Sperry Rand Corporation. The ALGOL compiler was constructed at the Case Institute of Technology by a team originally headed by Joseph Speroni (now at UNIVAC International). The input-output and backing store facilities, together with facilities for string handling and other valuable extensions to ALGOL 60, have been carried over to the SIMULA system. The authors are indebted to Nicholas Hubacker (UNIVAC International) for help in finding the way through the ALGOL compiler.

The authors wish to express their sincere thanks to those who have contributed to the development of the SIMULA project, in particular to: Bernard Hausner, of Rand Corp., whose experience with SIMSCRIPT had no little impact at an early stage; Ken Walter, of Case Institute of Technology, who boosted the project during a few summer months; Björn Myhrhaug, whose contribution to the implementation has been indispensable; and Sigurd Kubosch, who is rapidly becoming indispensable. Dag Belsnes and Sigurd Kubosch have given valuable comments and criticism concerning the form of the present paper. The authors also wish to thank the Programming Languages editor of this journal for his assistance.

RECEIVED OCTOBER, 1965; REVISED MAY, 1966

REFERENCES

1. NAUR, P. (Ed.). Revised report on the algorithmic language ALGOL 60. *Comm. ACM 6* (Jan. 1963), 1–17.
2. DAHL, O.-J., AND NYGAARD, K. SIMULA—a language for programming and description of discrete event systems, introduction and user's manual. Norwegian Computing Center, Forskningsveien 1 B, Oslo 3, Norway, May 1965.
3. HOARE, C. A. R., AND WIRTH, N. A. Contribution to the development of ALGOL 60. *Comm. ACM 9* (June, 1966), 413–432.
4. CONWAY, M. E. Design of a separable transition-diagram compiler. *Comm. ACM 6* (July 1963), 396.
5. WEIZENBAUM, J. Symmetric list processor. *Comm. ACM 6* (Sept. 1963), 524–544.
6. WEIZENBAUM, J. Letter to the editor. *Comm. ACM 7* (Jan. 1964), 38.