SIMULA information

NORSK REGNESENTRAL

NORWEGIAN COMPUTING CENTER
OSLO 3 - NORWAY

SIMULA™ 67

AN ADVANCED PROGRAMMING
AND SIMULATION LANGUAGE

JACOB PALME
(TRANSLATED BY BJØRN MYHRHAUG)

NORSK REGNESENTRAL / NORWEGIAN COMPUTING CENTER

Forskningsvn. 1 B, Blindern, Oslo 3, Norway, telefon (02) 46 69 30

Publication No. S-17

May 1970

SIMULA[TM] 67

An advanced programming and simulation language

Jacob Palme

(Translated by Bjørn Myhrhaug)

This report is an English translation of report C 8242-64(11) of the Swedish Research Institute of National Defence (FOA) published with their kind permission by NCC.

· SIMULA is registered trademark of the Norwegian Computing Center

© Copyright 1970 Jacob Palme

Summary

For programming simulation models there are special languages called simulation languages. Such languages usually have great advantages in that they give a clearer program and greater security against programming errors, and that less time is spent in actual programming and program modifications.

SIMULA 67 is one of the most advanced simulation languages. In the author's opinion, it has great advantages over other simulation languages, especially for the kind of simulation models used at FOA. A particular advantage of SIMULA 67 is that it is a general purpose programming language based on ALGOL 60, which makes SIMULA 67 equally suitable for programs which, besides the simulation part, also contain numerical calculations, symbol manipulations, list structures and text handling.

The report contains a comparison of some simulation languages of interest, a presentation of their characteristics (particularly the characteristics of SIMULA 67) and a brief introduction to some important concepts of SIMULA 67. The report does not contain a complete description of SIMULA 67.

Contents

1. Introduction ---------- 4
2. Requirements for a good programming language ---- 5
3. Doctor's reception, a simple example ---------- 7
4. Time-sequencing, event-sequencing or quasiparallel execution ---------- 9
5. List structures ---------- 14
6. Some important concepts of simulation languages - 18
7. Comparison of simulation languages ---------- 19
8. Reasons for choice of SIMULA 67 ---------- 21
9. The class concept of SIMULA 67 ---------- 23
10. Example of a SIMULA 67 program ---------- 25
11. References ---------- 27

1. Introduction

This report is intended to

a. give a short presentation of the most important characteristics of some well-known simulation languages.

b. present reasons why the author considers that FOA should choose SIMULA 67 for their simulation models.

c. give a short presentation of the most important concepts of SIMULA 67.

The special problems arising from discrete-event simulation of continuous processes of the differential equation type are not discussed in this report.

2. Requirements for a good programming language

These requirements have been presented in greater detail in an earlier report: "What is a good programming language?" (in English) by Jacob Palme, FOA P Report C 8231-64(11). A short summary is given below:

A good programming language ought to have the following characteristics:

Generality means that one and the same language can be used for many different applications. This makes it easier to write programs comprising different problems of different kinds. Also, programmers need not learn so many different languages.

Simplicity means that the language has a well thought out, simple but general basic structure, since this simplifies the learning and understanding of the language and its structure. The number of concepts in the language should be few, but those included should be powerful.

Extendability may offer a solution to the problem of combining generality and simplicity. An extendable language has a simple, general base and means for the user to extend the language to a special purpose language to suit his own requirements. The drawback with extendable languages is the difficulty of implementing them with reasonable efficiency. Simpler, more restricted extension facilities may, however, be implemented without this difficulty.

Model closeness means that the user can write a program which gives a good picture of his intuitive model of the problem. A particular case of model closeness is readability - the possibility to easily read and understand a finished program. Programs with a higher degree of readability are usually called self-documenting programs, i.e. they are so readable that they need little or no documentation except the program listings themselves.

Program structuring means that the user can easily divide his programs into separate parts, and integrate these parts into an easily understandable program structure.

Security means that the language is designed in such a way that it is possible for the compiler and the runtime system to detect as many programming errors as possible at an early stage.

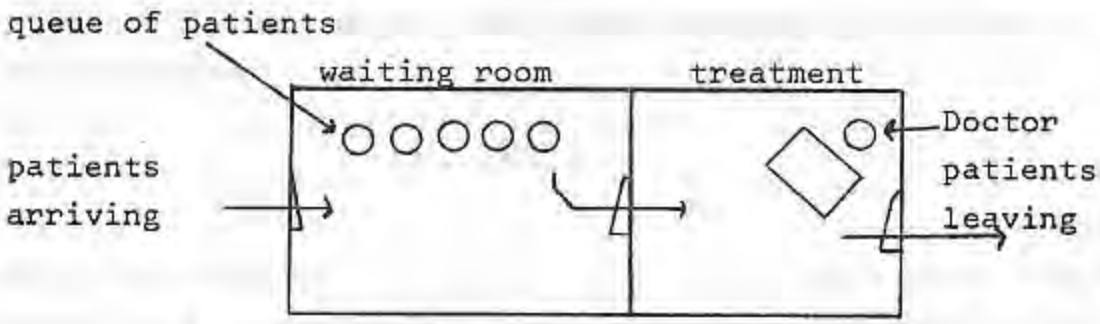
Efficiency means that the language is designed in such a way that it is possible to write a compiler which is fast, and to produce a program which does not waste computer time and other hardware facilities.

Compatibility means that the language is machine independent, so that a program may easily be transferred from one computer to another.

Special requirements - a basic requirement for a good general purpose programming language is good facilities for the following program methods: numerical calculations, string handling, list structures, input-output, error tracing. From FOA's point of view, simulation capabilities are also a desirable requirement.

3. Doctor's reception, a simple example

In this report, a simplified model of a doctor's reception will be used to illustrate different programming methods. The system is as follows:



When the doctor's reception opens, at time zero, there are KSTART patients in the queue outside the door. The patients are then admitted promptly at time interval TPA until time TSLUT, when the reception is closed. After closing the reception, the doctor treats the patients remaining in the waiting room before he leaves.

The time taken by the doctor to treat one patient varies randomly. During each time interval the doctor is treating a patient, we assume that the probability is equally large of his completing the treatment in that time interval.

In each time interval $(u, u+\Delta t)$, while the doctor is treating a patient, the probability that he will be finished with that patient within the time interval is $a \cdot \Delta t$. The probability of the treatment taking longer than time t is thus:

$$P(T>t) = \lim_{\Delta t \to 0} (1-a\Delta t)^{\frac{t}{\Delta t}} = e^{-at}$$

The treatment time for the patient has thus the probability distribution

$$F(t) = P(T<t) = 1 - e^{-at}; \quad 0<t$$

and the frequency distribution is

$$f(t)\,dt = dF(t) = a\,e^{-at}\,dt$$

From a random number s, with rectangular distribution in the interval (0,1), one may generate a random number, b, with the distribution given above through the inverse of the distribution:

$$b = \frac{-\log s}{a}$$

Since the doctor's reception is a very simple model, it does not give a true picture of the difference in complexity between programs written in different ways and in different programming languages.

But a doctor's reception is a typical queue model. Models which are not typical queue-type models are often more difficult to program, because many simulation languages are specially oriented towards queue models. The choice of a doctor's reception as an example was taken because the model is so simple and well-known that it gives a clear picture of the functions of different programming languages and methods.

4. Time-sequencing, event-sequencing or quasiparallel execution

A simulation program is usually written according to one of the following three main principles. More than one principle may, however, be applied within a program. The principles are shown by the examples in figures 4.1, 4.2 and 4.3.

Time-sequencing means that the time is advanced by a fixed time step, and that model-descriptive data is updated for each time step. Time-sequencing simplifies programming of models which to a large extent consist of complex systems of differential equations. The drawbacks are long execution times and the difficulty of avoiding systematic errors in the results. See figure 4.1.

Event-sequencing means that the real continuous process is divided into a number of instantaneous events. The events are executed in ascending order of time, and during each event model data is updated and possibly new events are generated. Programming languages for event-sequencing are usually capable of handling time-sequencing problems. See figure 4.2. Event-sequencing is the main principle of SIMSCRIPT.

Quasi-parallel sequencing means that each physical object (e.g. aeroplane, anti-aircraft gun, doctor, patient, crane, car) is described by one program, commonly known as a "process". The programs for the different entities are executed in quasi-parallel, in that the computer partly works on one, partly on another process as the simulated time increases. Quasi-parallel sequencing is, in reality, a generalization of event sequencing or time sequencing in which each process incorporates more than one event. The advantage of quasi-parallel sequencing is that the programs are readable and permit a high degree of model closeness. A process gives a natural picture of the object it describes together with everything relevant to the object (actions, data, special procedures) collected in one place in the

program. Programming languages for quasi-parallel execution can also be used for event-sequencing and time-sequencing. See figure 4.3. Quasi-parallel execution is the main principle in SIMULA.

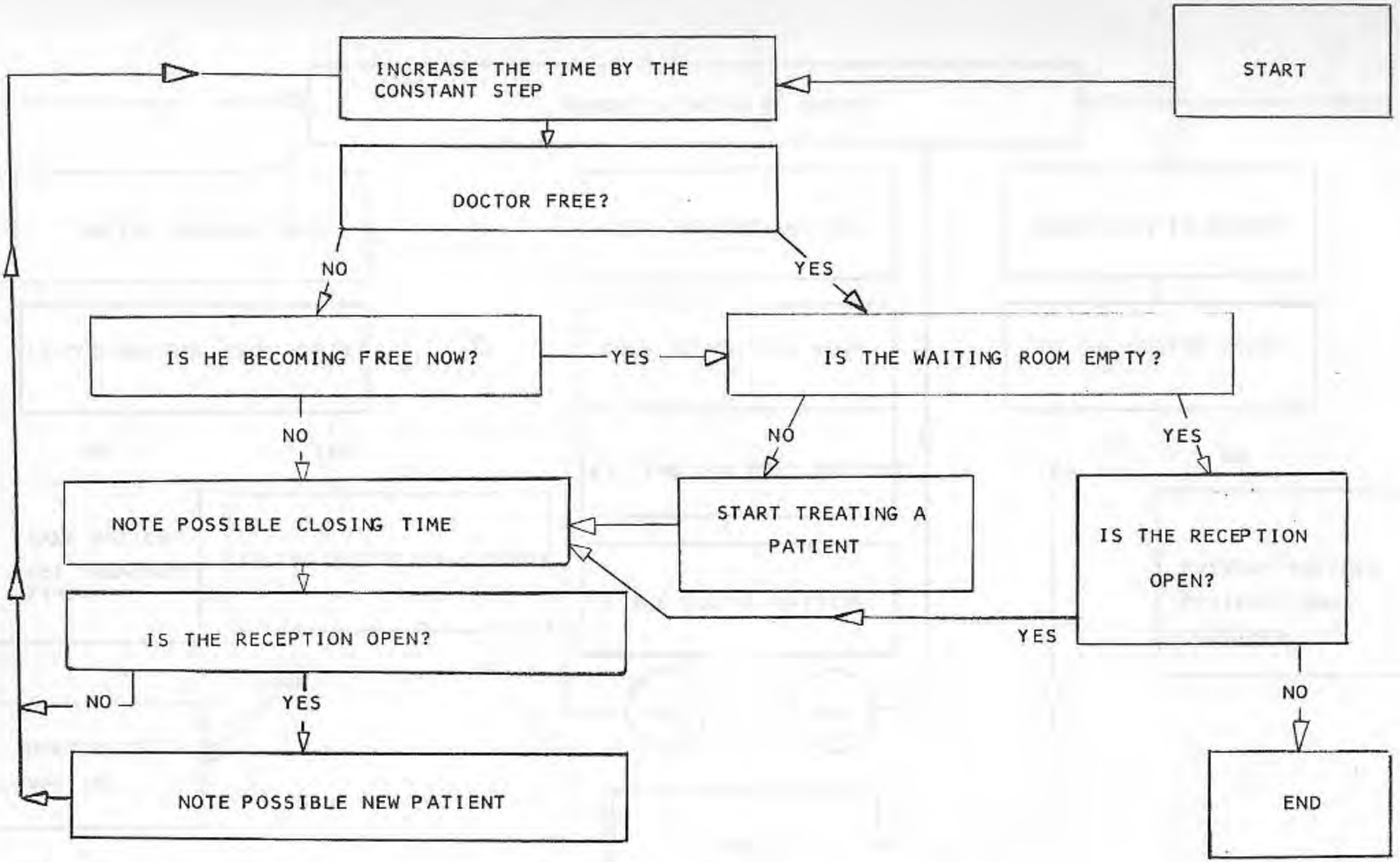


FIG 4.1 TIME-SEQUENCED VERSION

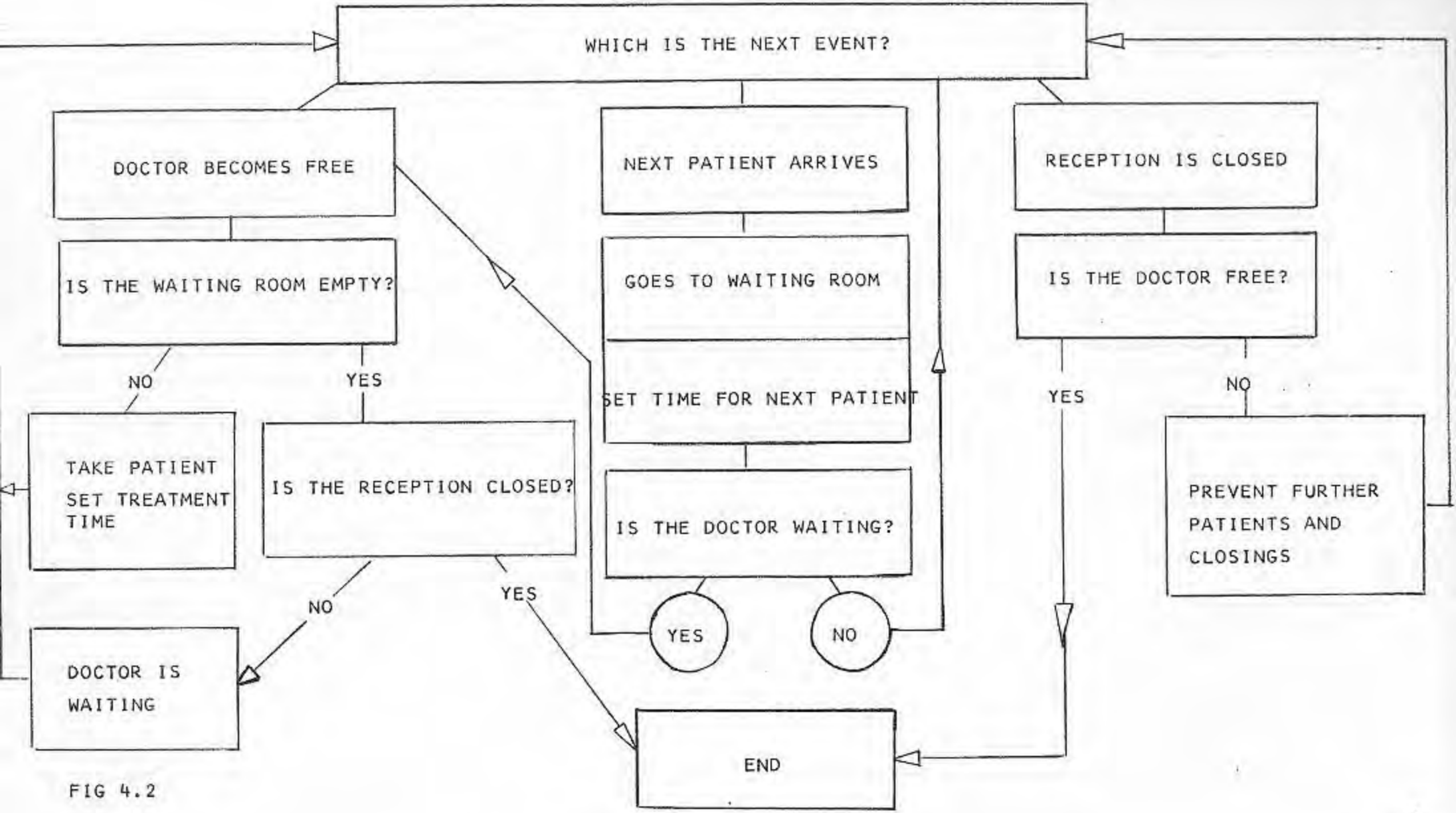


FIG 4.2

EVENT-SEQUENCED VERSION

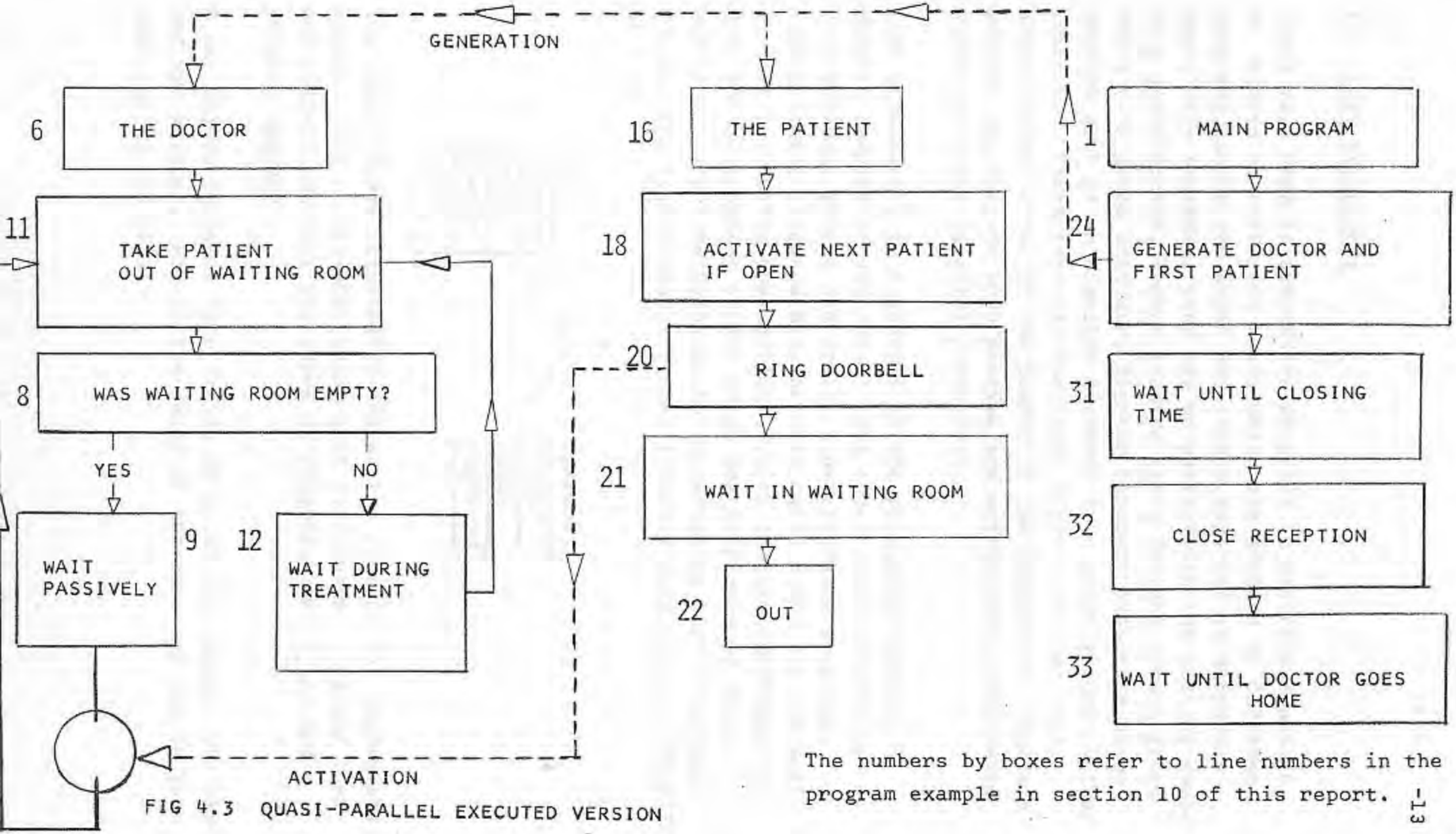


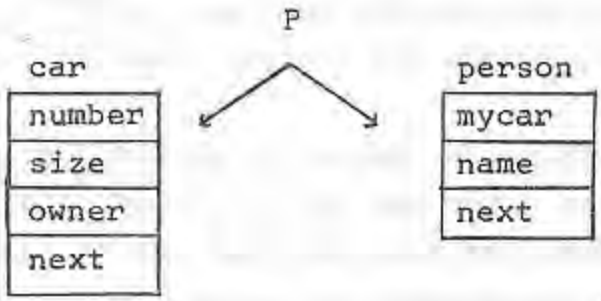
FIG 4.3 QUASI-PARALLEL EXECUTED VERSION

The numbers by boxes refer to line numbers in the program example in section 10 of this report.

5. List structures

There is a need in simulation programs to describe a number of objects of different kinds which are members of different sets and queues and which are linked together in different ways. For instance, they may be patients in the waiting queue of a given doctor, trucks loading given freight from a given crane to a given customer, fighter aircraft attacking given bombers with given missiles directed from given ground control-stations. Characteristic for these objects is that their number often varies in the course of the simulation. Certain objects may include both passive and active data, executable events or quasi-parallel processes.

List structuring is a general method frequently used to program complex structures of this type. List structures give general, simple and easily understandable programs. A programming language may, however, give inadequate security in list processing applications, which in turn increases the risk for programming errors which are difficult to detect. SIMULA 67 uses a methodology for programming list structures which gives significantly better security than either SIMSCRIPT or PL/I.



The figure above illustrates one of the reasons why inadequate security may result when using list structures. Assume that our program contains two types of objects, cars (car) and persons (person).

A car object contains four fields of data: its number, its size and two pointers, one to the owner of the car, and one to the next car in the file.

A person object contains three fields of data, mycar pointing to the car object belonging to this person, the persons name, and a pointer to the next person in the file.

Suppose we have a pointer variable, p, in our program which may sometimes refer to car-objects, sometimes to person-objects. A programming error which commonly occurs is that one believes that p is referring to a person-object when it is actually referring to a car-object.

If this error has been made, then on attempting to locate the car (mycar) of this person, the number (number) will be obtained instead. When one later continues and looks at this car, one finds that instead one is looking in the computer memory at the location which has the number of the car as its address. Disconcerting errors which are not easily detectable will result.

Because of inadequate security, a similar kind of error may occur when an object is de-allocated while there are still pointers referring to it.

PL/I and SIMSCRIPT both contain these forms of inadequate security, but they have been eliminated in ALGOL 68 and SIMULA 67. The basic method for ensuring adequate security is:

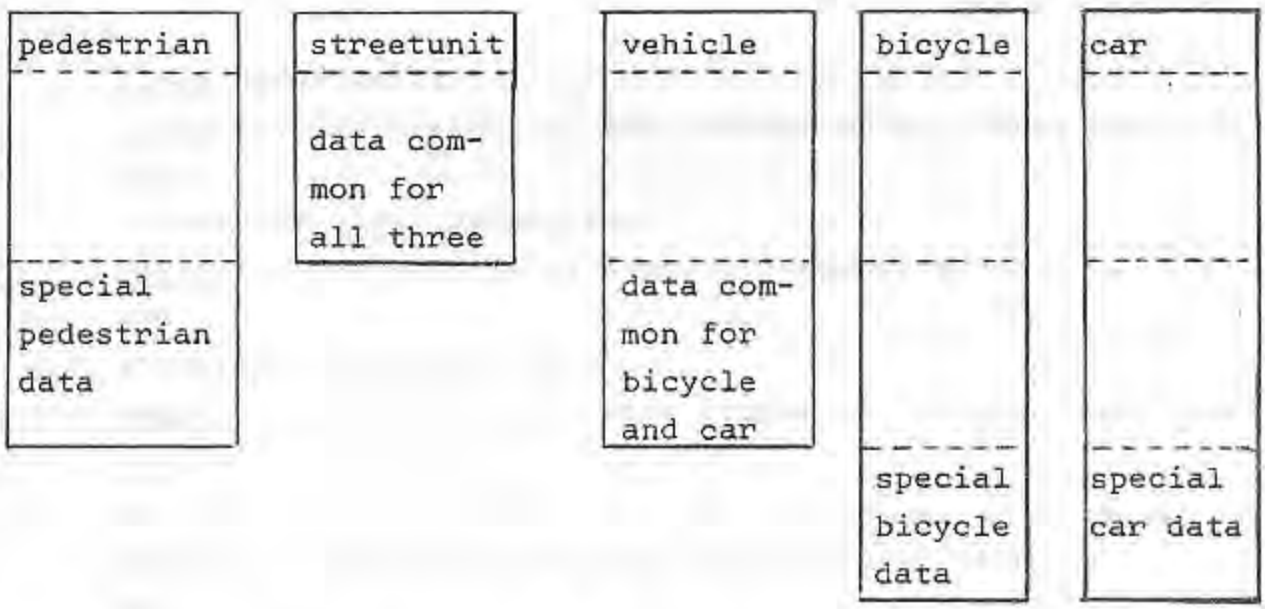
a) When a pointer is declared, one must specify the kind of objects this pointer may refer to.
The pointer may only be used to refer to this type of object. Control may be carried out during compilation, and is therefore cheap.

b) The programmer may not explicitly de-allocate objects. Instead they are automatically de-allocated by a "garbage collector" when no accessible pointers refer to the object.

The drawback with this method of obtaining security is that the programmer sometimes needs to allow one pointer to refer to different kinds of objects, for instance, to be able to handle easily queues containing different kinds of objects.

Two methods have been proposed to solve this difficulty. One method, used in ALGOL 68, allows a pointer to refer to several objects from a specified list of permissible kinds of objects. One drawback is that this is very inefficient, since every reference must be checked for correct usage at run time.

Another solution is to have hierarchical structures of objects. Suppose that we want to have three kinds of objects, pedestrian, bicycle and car. We may then define the following five objects:



When a pointer is declared, it is limited to one of the five units above. However, a pointer which is limited to a vehicle, for example, may also refer to all units which include vehicle, i.e. bicycle and car. It may, however, only access vehicle-data in these objects, not the special bicycle and car-data. If access to these is desired, one must first explicitly test whether the pointer is referring to a bicycle or to a car.

Thus one may select deep pointers, such as bicycle-pointers, which may access all bicycle data, but nothing other than bicycle data, or shallow pointers, such as streetunit-pointers, which may access all five kinds of objects, but only the common data.

This solution gives complete security, the possibility to carry out all implicit checking during compilation, and almost as large flexibility as in SIMSCRIPT or PL/I.

The solution corresponds well with the requirement for a realistic model, since such a hierarchical or generic structure is a natural way of describing reality in many cases.

To illustrate SIMULA 67, we give below an extract of the declaration of the above objects:

```
begin
    class streetunit;
    begin ... declaration of data common to all three units ...
    end;
    streetunit class pedestrian;
    begin ... declaration of special "pedestrian"data ...
    end;
    streetunit class vehicle;
    begin ... declaration of data common to "bicycle" and "car" ..
    end;
    vehicle class bicycle;
    begin ... declaration of special "bicycle" data ...
    end;
    vehicle class car;
    begin ... declaration of special "car" data ...
    end;
    ... and here comes the program which uses the units
        declared above ...
end;
```

6. Some important concepts of simulation languages

Imperative sequencing: the program directs when different events are to occur.

Interrogative sequencing: certain events occur automatically when given conditions are fulfilled. This is convenient, but often yields inefficient programs.

External sequencing: certain events are generated from without, through an input medium, such as a data set or display unit. A programming language which does not have external sequencing may achieve this through a routine which communicates with the input unit and sequences imperatively.

Dynamic object: object in a list structure, having the characteristic which enables it to be both generated and eliminated from the model dynamically during the simulation.

De-allocation: possibility for the programmer to explicitly eliminate dynamic objects. De-allocation often results in inadequate security, since references may still exist for an object even after it has been de-allocated.

Sets and queues: unordered or ordered collections of units, where the number of units in the collection may increase or decrease during the simulation.

7. Comparison of simulation languages

Some of the points of view expressed in this table are subjective personal judgements. Another person may have quite another point of view. One language may often have certain characteristics which are good from one point of view, another language may have different characteristics which are good from the same point of view. The weighing up of such varied characteristics is a very difficult task.

Characteristic	GPSS	SIMSCRIPT II	SIMULA 67
Base language	None in excess of assembler	Fortran-like and PL/I-like	ALGOL extended by e.g. text handling
Basic concepts	Flow of units through block-diagram	Event-sequencing and list struct.	Quasi-parallel execution and list struct.
Sequencing	Imperative and interrogative	Imperative and external	Imperative
Scale of time measurement	Integer	Real	Real
Reference to dynamic objects	Automatic, hidden from programmer	With pointers, sets and queues	With pointers, sets and queues
De-allocation of dynamic objects	Programmed	Programmed	Automatic
Security	Fairly good	Low	Good
Sets and queues	Implicit	Several different types	One general type
Readability	Very poor	Good	Good
Program structuring	Limited	Fairly good	Very good
Can things belonging together be collected at one place in the program	To a certain extent	To a certain extent	Very good

Standard elements for collection of statistics and giving results	Very good standard statistics but difficult to get results other than standard	Very Good	Good
Model closeness	Good for queue models, very bad for other models	Fairly Good	Very Good
Possibility to extend the language to users own models - special language for special purposes	Does not exist	Limited	Fairly good
Efficiency	Very low	Good	Good
Limitations	Only useful for simple queue models	Few	Few

8. Reasons for the choice of SIMULA 67

Generality SIMULA 67 is a general purpose language which may be used for most programming tasks, not exclusively for simulation. There are also good possibilities for the use of FORTRAN-programs as subroutines to SIMULA 67 programs.

Simplicity SIMULA 67 is built around few basic concepts. Experience shows that SIMULA 67 causes certain problems for the beginner initially, until the consequences of the quasi-parallel execution concept and the class concept have been understood. The language is easy to use once these initial problems have been overcome.

Extendability SIMULA 67 has a limited, but simple method of extending the language, making it a special-purpose language for special applications. The method is more limited than, for example, in ALGOL 68 or GPL, but it is simpler to learn and easier to implement efficiently. See section 9 of this report.

Model closeness Quasi-parallel execution and the possibility to group data, process and special procedures belonging to a given object together in the program give very good model closeness for most simulation applications. SIMULA 67 also has possibilities to use the extension mechanism to make the language more "modelclose" to special models, such as diagram processing or simulation of digital circuits.

Readability Readability is good, and it is easy to write self-documenting programs.

Program structuring is very good through program parentheses of the begin-end type, and through the procedure and class concepts.

Security is good, especially with list structures. Experience shows that these usually cause errors which are difficult to find if the system has low security.

Efficiency is good. It depends, however, on the amount of work spent in making compilers efficient, and here the IBM-supported languages have usually an advantage due to the larger resources which IBM can use in compiler development.

Compatibility is good, since the language is implemented on many computers and has a strict definition.

Text processing and input-output are defined in SIMULA 67.

9. The class concept of SIMULA 67

The two most important new concepts of SIMULA 67 are quasi-parallel execution and the class concept. Quasi-parallel execution has been described in section 4 of this report.

The class concept implies that one may describe one concept which at the same time may function as:

a) a dynamic object in list structures. SIMULA 67 has a unique method of achieving high security without restricting the programmer. See section 5 of this report.

b) a quasi-parallel process. See section 4 of this report.

c) local, object-bound procedures. These may be accessed through pointers to the class. Examples of such procedures: If the class describes a line in a diagram processing program, the procedures may be used to move the line in the diagram to compute distances to other objects in the diagram etc.

d) a hierarchical structure. A class may be an extension of another class, and then have all plus something more of the previous class characteristics. A third class may subsequently be an extension of the second etc.

The class concept may be used to extend SIMULA 67 to a special purpose language for special applications, such as diagram processing or tank battle simulation.
An example is given below of parts of such an extension to a special application on automatic justice (!)

```
BEGIN
     REF (CARACCIDENT) HITANDRUN;
     CLASS CARACCIDENT;
     BEGIN
          REF (PERSON) PROCEDURE GUILTY;
          BEGIN COMMENT FINDS WHO IS GUILTY IN THIS CAR ACCIDENT;
          END;
     END;
     CLASS PERSON;
     BEGIN
          REF (DRIVINGLICENCE) LICENCE;
     END;
     CLASS DRIVINGLICENCE;
     BEGIN
          PROCEDURE SUSPEND (SUSPENSIONTIME);
             REAL SUSPENSIONTIME;
          BEGIN COMMENT SUSPENDS THE LICENCE FOR THE
          SUSPENSION TIME;
          END;
     END;
```

COMMENT After declaring the classes and procedures from which extracts are given above, one may use the extended language for instance in the following statement:

```
HITANDRUN.GUILTY.LICENCE.SUSPEND(3*YEAR);
```

COMMENT This statement means:

a) HITANDRUN refers to a CARACCIDENT

b) Call the version of the procedure GUILTY associated with this CARACCIDENT

c) The procedure gives as a result a pointer to the PERSON who is guilty

d) This PERSON has a pointer LICENCE local to himself which refers to his drivers licence.

e) Call the version of the procedure SUSPEND associated with this drivers licence. As parameter is given "3 YEAR", meaning three years.

f) This procedure will suspend the drivers licence in question for three years.

```
END;
```

10. Example of a SIMULA 67 program

(Not tested on a computer)

```
1.  SIMULATION BEGIN                    Simulation of doctors reception
2.  INTEGER I, KSTART, U; REAL A, TPA, TSLUT;
3.  REF(HEAD) WAITROOM;                 "WAITROOM" is a queue of
                                        waiting patients
4.  REF (DOCTOR) DOORBELL;              "DOORBELL" is a pointer to the
                                        doctor
5.  BOOLEAN OPEN;                       "OPEN" indicates if the
                                        reception is open
6.  PROCESS CLASS DOCTOR;               "DOCTOR" describes the doctors
                                        actions
7.          BEGIN
8.          NY: IF WAITROOM.EMPTY       The doctor becomes passive if
                                        no patient is waiting
9.              THEN PASSIVATE
                ELSE
10.             BEGIN                   If patient(s) are present,
                                        take the first one from the
11.             WAITROOM.FIRST.OUT;     waiting room.  The doctor
                                        becomes busy for an exponent-
12.             HOLD(NEGEXP(A,U))       ially distributed time interval
13.             END;
14.             GO TO NY;               When the doctor becomes free,
                                        he repeats his working cycle
15.         END;
16. PROCESS CLASS PATIENT;              "PATIENT" describes a patient's
                                        actions.  For each new patient,
17.         BEGIN                       the system generates a new
                                        "PATIENT"
18.         IF OPEN THEN                The patient generates the next
                                        one as long as the reception
19.            ACTIVATE NEW             remains open
               PATIENT DELAY TPA;
20.         ACTIVATE DOORBELL           The patient reminds the doctor
               DELAY 0;                 about his arrival
21.         WAIT(WAITROOM);             The patient waits in the waiting
                                        room
22.         END;
```

```
23.  WAITROOM:-NEW HEAD;                    The waiting room is generated
24.  DOORBELL:-NEW DOCTOR;                  The doctor is generated
25.  U:=ININT; KSTART:=ININT;
     A:=INREAL;
     TPA:=INREAL;
     PSLUT:=INREAL;
26.  FOR I:= 1 STEP 1 UNTIL
             KSTART DO                      "KSTART" patients arrive before
                                            reception is opened
27.  ACTIVATE NEW PATIENT;
28.  OPEN:=TRUE;                            The reception is opened
29.  ACTIVATE DOORBELL;                     The doctor arrives
30.  ACTIVATE NEW PATIENT DELAY TPA;        The first patient after
                                            opening is generated
31.  HOLD(TSLUT-TPA);                       The main program waits until
                                            closing time
32.  OPEN:=FALSE;                           Reception is closed
33.  HOLD(10000*TSLUT);                     Main program is ended
34.  END;
```

11. References

Birtwistle, G. 1969: Notes on the Simula Language, The Norwegian Computing Center, Forskningsveien 1 B, Oslo 3, Norway.

Birtwistle, G. 1969: Notes on Simula System Classe. The Norwegian Computing Center, see above.

Dahl, O.J. Nygaard, K and Myhrhaug, B. 1968: SIMULA 67. Basic Information. The Norwegian Computing Center, see above.

Dahl, O.J, Nygaard, K and Myhrhaug, B. 1967: SIMULA 67 Common Base Language. The Norwegian Computing Center, see above.

Dahl, O.J. 1966,68: Discrete Event Simulation Languages. Programming languages. NATO Advanced Study Institute. Dec. 1968.

Kiviat, P.J, Villanueva, R and Markowitz, H.M. 1968: The Simscript II Programming Language, Rand Corp, 1700 Main Street, Santa Monica, California. U.S.A.

Kiviat, Philip J, 1969: Digital Computer Simulation: Computer Programming Languages, Rand Corp, see above.

Markowitz, Hausner B and Karr H.W., 1963: SIMSCRIPT, A simulation programming language, Prentice Hall 1963

Palme, J. 1968: Are Simulation Languages Convenient for Military Simulations? FOA P Rapport C 8210

Palme, J and Enderin, L. 1968: SIMULA 67 ett språk för framtiden. Modern Datateknik, 9, 1968.

Palme, J. 1969: What is a good programming language? FOA P Rapport C 8231