SIMULA

A language for programming and
description of discrete event systems.

Introduction and user's manual

BY

OLE-JOHAN DAHL AND
KRISTEN NYGAARD
5th EDITION SEPTEMBER 1967
NORWEGIAN COMPUTING CENTER
FORSKNINGSVEIEN 1 B
OSLO 3 - NORWAY

- SIMULA -
A LANGUAGE FOR PROGRAMMING AND
DESCRIPTION OF DISCRETE EVENT SYSTEMS
INTRODUCTION AND USER'S MANUAL
BY
OLE-JOHAN DAHL AND KRISTEN NYGAARD
# Table of Contents

**Preface**

**Part I**  
**Introduction**

**Chapter 1**  
Simulation and Discrete Event Systems

- 1.1 The SIMULA Project  
- 1.2 Simulation  
- 1.3 SIMULA Design Objectives  
- 1.4 Discrete Event Systems

**Part II**  
**The SIMULA Language**

**Chapter 2**  
Processes

- 2.1 Basic Properties  
- 2.2 The Sequence Control  
- 2.3 States  
- 2.4 Exogenous Attributes

**Chapter 3**  
Elements and Sets

- 3.1 Basic Concepts  
- 3.2 Elements and Element Variables  
- 3.3 Sets  
- 3.4 Element Expressions  
  - 3.4.1 Generative expressions  
  - 3.4.2 Set membership references  
- 3.5 Boolean Expressions  
- 3.6 Element Operations  
- 3.7 Non-elementary Procedures  
- 3.8 Examples

**Chapter 4**  
Sequencing

- 4.1 The Sequencing Set  
- 4.2 Sequencing Statements  
- 4.3 Scheduling Statements  
- 4.4 SQS Functions  
- 4.5 Examples
PART III SYSTEM DESCRIPTION

CHAPTER 11 ALGOL FUNDAMENTALS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 Simple Variables and Declarations</td>
<td>75</td>
</tr>
<tr>
<td>11.2 Statements and Programs</td>
<td>76</td>
</tr>
<tr>
<td>11.3 Compound Statements</td>
<td>77</td>
</tr>
<tr>
<td>11.4 Labels and go to - statements</td>
<td>78</td>
</tr>
<tr>
<td>11.5 if - then - statements</td>
<td>79</td>
</tr>
<tr>
<td>11.6 Arrays</td>
<td>81</td>
</tr>
<tr>
<td>11.7 for - statements</td>
<td>82</td>
</tr>
<tr>
<td>11.8 Blocks</td>
<td>83</td>
</tr>
<tr>
<td>11.9 ALGOL Programs</td>
<td>85</td>
</tr>
<tr>
<td>11.10 Procedures</td>
<td>85</td>
</tr>
</tbody>
</table>

CHAPTER 12 A SIMPLE SIMULA DESCRIPTION

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1 A Simple Disease System</td>
<td>89</td>
</tr>
<tr>
<td>12.2 Details of the Element and Sequencing Procedures</td>
<td>97</td>
</tr>
<tr>
<td>12.3 Details of Scanning and Connection</td>
<td>105</td>
</tr>
</tbody>
</table>

CHAPTER 13 A WORKED EXAMPLE

Index of Library procedures 122
Preface to the 5th Edition

This report presents the SIMULA language. The first part, the chapter INTRODUCTION, contains a brief outline of the basic approach to system description and simulation reflected in the language. The second part, THE SIMULA LANGUAGE, gives the language definition and serves as a user's manual.

No comprehensive SIMULA textbook has been written, but two examples and some basic ALGOL 60 information are contained in part III.

SIMULA has been in increasing use at UNIVAC 1107 and 1108 installations since the beginning of 1965, and a revised version of "Report on the Use of SIMULA" is now being written, giving a survey of SIMULA jobs and application areas. The "SIMULA Tracing and Debugging System" is described in a separate report, and another report will be issued describing the procedures for giving output to the "KINGMATIC" drawing machine.

The changes from the first to the fifth edition of this manual consist mainly of the introduction of additional procedures and a worked example.

The authors have recently developed a new general programming language, named SIMULA 67. ("SIMULA 67 Common Base Definition", June 1967.) This language, now being implemented on various computers, is a major extension of the SIMULA presented in this manual. SIMULA 67 has also extended simulation capabilities over the present SIMULA.
Some of those who have contributed to SIMULA in various ways are mentioned in the introduction. We should also like to thank UNIVAC staff for assistance, particularly Nicholas Hubacker and Joseph Speroni who helped us to find our way through the UNIVAC 1107 ALGOL compiler.

Björn Myhrhaug and Sigurd Kubosch have been members of our team for a long time, and their contributions have been indispensable.

Oslo, September 1967
CHAPTER 1.

SIMULATION AND DISCRETE EVENT SYSTEMS.

1.1 The SIMULA Project.

The two main objects of the SIMULA language are:

1. to provide a language for a precise and standardized description of a wide class of phenomena, belonging to what we may call "discrete event systems".

2. to provide a programming language for an easy generation of simulation programs for "discrete event systems".

SIMULA is based on ALGOL and contains that language as a subset. The extension consists of the introduction of some new basic concepts, new statements which operate upon these concepts and a set of library procedures. In order to achieve the greatest possible logical integration of SIMULA and ALGOL, the scope of some of the ALGOL statements has been extended to operate also upon the new concepts introduced.

SIMULA has been developed by the authors at the Norwegian Computing Center, since the summer of 1963 under a contract with UNIVAC Division of Sperry Rand Corporation. A SIMULA compiler for the UNIVAC 1107 and 1108 Computers is completed and has been in operation since December 1964.

In addition to the authors Björn Myhrhaug at the NCC, Bernard Hausner (now at RAND Corporation) and Ken Walter (now at Case Institute of Technology) have taken part in various stages of the project.
The authors also want to express their gratitude to all the programmers and operational research workers who have taken interest in the project and contributed useful ideas and comments.

1.2 Simulation.

Before the appearance of the electronic computer, properties of telephone communication systems, ticket counter systems, machine systems, etc. were mainly analyzed by analytic means.

In mathematics the theory of stochastic processes furnished the basic tools for work in these areas. Important and interesting results were, and still are obtained by this approach.

The technological development and the introduction of scientific methods for planning and decision making has created a need for study of complex systems. The usefulness of the analytical approach is, however, limited to rather simple situations. The electronic computer has made simulation an extremely powerful method of analysis, and the use of simulation has been extended to many other areas than those mentioned above by new disciplines like operational research. By simulation it is possible to study very complex problems, and both transient and stationary states of systems may be analyzed whereas the analytical approach often is limited to stationary states.

This does not imply that analytical studies have become obsolete. If a solution is obtainable from a realistic analytical model, it offers more complete and reliable information than the statistical inference from a simulated sample of system runs.

Since it is possible to simulate very complex situations, it is tempting to substitute a "too realistic" simulation for clear thinking and valid simplification. This is done by the introduction of so many variables and variations in decision
rules that the basic structure of the problem is obscured and no certain information on the important features of the system is obtained.

Also, one is tempted to forget that in many situations (e.g. in military operational gaming) the number of available combinations of strategies is so large that only a very small percentage may be simulated.

The programming of a simulation may be very time-consuming. This is not too serious if a large number of alternatives are to be run and the model of the actual system and the decision rules to be used are well defined.

This is as a rule not the case. On the contrary, one wants to experiment with different layouts and decision rules, trying to understand the system, gradually introducing more complexity in those parts of the system where this is essential. Very often it is found that apparently minor changes in the system call for extensive reprogramming, and the usefulness of the approach is greatly reduced.

The advent of the algorithmic languages like ALGOL and FORTRAN has only slightly improved the situation. These languages are basically operating on fixed data structures and have simpler principles for sequencing of actions than needed in simulation.

General simulation programs have been developed. Some are rather ad hoc tools for faster generation of programs, other try to create simulation languages. The best known languages are GSP by Tocher and his colleagues, CSL by Buxton and Laski and SIMSCRIPT by Markowitz, Hausner and Karr. The authors have gained from their acquaintance with these efforts.
SIMPAC by Lackner was not available to the authors. GPSS by Gordon is so different from SIMULA in its approach that it has not had much influence on the latter.

The last, but perhaps the most important restriction in the use of simulation is the lack of a basic language for problem formulation. Programming cannot be started before the system is precisely described. There is a strong demand for basic concepts useful in understanding and describing all systems studied by simulation from a common point of view, for standardized notation, for easy communication between research workers in different disciplines.

SIMULA is an attempt to meet this demand. Problem formulation and not program generation has been our starting point. This being said, it is clear that the language should be designed so that system descriptions may produce simulation programs through a compiler.

1.3 SIMULA Design Objectives.

1. Since simulation is the method of analysis most commonly to be used for the systems in which we are interested, SIMULA is a dynamic language:

   It is designed to describe sequences of actions, not permanent relationships. The range of variation in decision rules and interactions between components of systems is so wide that it is necessary to let the language contain a general algorithmic language. An important reason why ALGOL has been chosen is that its block structure is similar to what was needed in SIMULA.

2. SIMULA should be built around a few basic concepts, selected to provide the research worker with a standardized approach to a very wide class of problems and to make it easy to identify the various components of the system.
3. Attempts have been made to make the language unifying - pointing out similarities and differences between systems, and directing - forcing the research worker to consider all relevant aspects of the systems. Efforts have also been made to make SIMULA descriptions easy to read and print and hence a useful tool for communication.

4. Taking the above objectives into account, SIMULA descriptions (supplemented by the necessary input, output and data analysis statements) should without any rewriting be able to produce simulation programs for electronic computers through compilers.

1.4 Discrete Event Systems.

In our discussion of language concepts we will start with a well-known system:

An office with a series of ticket counters, offering a range of services to customers. Customers arrive in the system with time intervals described by an arrival distribution, they enter a queue and they are given service by a clerk behind a counter after some time in the queue. Then the customer disappears from the system or enters another queue.

The customers are moving through the system, the clerks remain. The queues also remain but their contents change.

One may regard the clerks as the active partners in the interaction taking place, pushing the passive customers from place to place.

The same point of view also is a natural one when studying a stream of materials through a factory. The materials are passive, the machines are active.
The customers (or the materials) may all be exactly similar, but often they must be characterized by priorities, by a description of the service they demand, their earlier history in the system etc. They are carriers of information.

This distinction between passive entities being carriers of information and active entities acting upon and pushing the passive ones around is so natural in many systems commonly studied that some simulation languages have introduced passive data carriers and active "machines", "stations" or "routines" as two basic concepts. This was also done in the early stages of SIMULA.

In social systems like epidemics, attitude diffusion etc., we encounter entities ("individuals") interacting with other entities of the same kind and at the same time acted upon by other entities (e.g. "treatments").

When we extend the counter system or the factory, it may be natural to regard the entities passive in one part of the system, as active in another part or in another situation. Also, it is always possible to reverse the point of view and describe the customers and the materials as the active partners, acquiring service from the passive clerks or machines only characterized by data describing their operational properties.

For these reasons, and to achieve greater flexibility and unity, SIMULA has integrated the two kinds of entities into one. The basic concept in SIMULA is the process, being characterized by a data structure and an operation rule.

The individual items of the data structure of a process will be called attributes.
A process may be active in some stages of its presence in the system, passive in others, depending upon its operation rule and interference with other processes. It is possible to let the operation rule of a process be rudimentary and in this way obtain an always passive data carrier as described above.

During an active phase of a process it may "connect" other processes, making their attributes available to itself for decision-making and manipulation. During an active phase a process also may "schedule" a later active phase for itself and other processes.

An active phase of a process is called an "event". The scheduling of an event is made by an "event notice" telling at which "system time" the event will occur and to which process it refers.

In SIMULA the system time is kept constant during an event, and hence all actions during an active phase may be regarded as instantaneous.

Thus SIMULA may be used to describe systems which satisfy the following requirement:

The system is such that it is possible to regard its operation as consisting of a sequence of instantaneous events, each event being an active phase of a process.

The number of processes may be constant or variable and they all belong to one or more classes called activities.
Since the set of system times at which events occur forms a discrete point set on the system time axis, and since every action in the system is a part of an event, we will name these systems **discrete event systems**.

By introducing suitable processes SIMULA also may be used to describe with the desired degree of accuracy continuously changing systems, as well as systems in which some processes have continuous changes, other processes discrete changes.

In addition to processes we need entities which may be used as "storing places" for processes: queues, files etc. The SIMULA "set" concept serves this purpose. The contents of a SIMULA set may consist of any mix of different kinds of processes and may be changed during the operation of the system.

We also want to be able to assign names to specific processes and to refer to processes through their event notices.

Because of the need for uniformity in the logical structure and the implementation on a computer, **all** references to processes in SIMULA are indirect, through a standardized reference called an "element".

To obtain a suitable system description, we may often need more than one element referring to the same process: The clerk at an airport check-in counter has to check if the passenger he is giving service also is referred to in the list of booked-in passengers. We may even want to have more references to the same process in a given set: In the police criminal records a person may be present under different aliases.
The naming of specific processes in SIMULA is achieved through "element variables" taking on specific elements as values.

Having introduced processes, event notices, sets, elements and element variables as basic concepts, we need statements utilizing and operating upon these concepts to be able to give a precise description of the data structures and operation rules of processes. Part II of this report gives precise definitions of the basic concepts and available statements in SIMULA.
CHAPTER 2.

PROCESSES

2.1 Basic Properties.

The process concept is fundamental in SIMULA. It serves to organize and classify the actions taking place in a discrete event system as well as the data involved. A discrete event system is viewed as a collection of processes, whose actions and interactions completely describe the operation of the system.

Processes enter and leave the system as the result of actions performed within the system itself. During its presence in the system a process must be regarded as an individual entity distinct from other processes. It has its own local data and its own behaviour pattern.

A system may contain several processes with a similar data structure and the same behaviour pattern. Such processes are said to belong to the same class, called an activity.

A process is described by an activity declaration, which is common to all processes belonging to that class. An activity declaration has the same syntax as a procedure declaration in ALGOL, except that the first symbol of the heading is "activity". There is no "<type> activity" concept. (See also section 2.4)

A reference to an activity is called a process designator (see section 3.4.1). It invokes the generation of an individual process described by that activity declaration, in which the formal parameters are replaced by the specified actual ones. The process designator is itself an expression referencing the generated process.
The process will usually remain in the system after the evaluation of the process designator. Its actions are executed in parallel with those of other processes in the system, in the sense that sequences of actions of different processes are interlaced.

The data structure belonging to a process are the data local to it, as defined in the activity declaration. The statements of the activity body constitute the operation rule of the process. The body can have an arbitrary block structure.

The items local to the outermost block of the body are called endogenous attributes; the formal parameters are called the exogenous attributes. The attributes of a process can be made accessible to the outside, i.e. to other processes, see chapter 5.

As the result of evaluating a process designator control enters the specified activity declaration. Thereby a local data block is created, which is distinct for this particular process. Two processes belonging to the same activity may differ in the values of their local variables, and they may be in different stages of execution at a given time.

2.2 The sequence control.

The actions performed by a process can be widely separated in time. They are grouped together in active phases separated by periods of inactivity. Inactive periods are invoked by certain sequencing statements, which are described in chapter 4. During such a period other processes are allowed to operate. However, during an active phase the process has complete control; there is no "interrupt" mechanism in the language.
A process can be viewed as a self-contained program having its own local sequence control. When a statement invoking an inactive period is executed, the local control will stay at the end of this statement during that period, and will proceed to the dynamically next statement at the time of the next active phase.

The main sequence control applying to a SIMULA program as a whole, will go from one process to another and execute one active phase at a time, in a sequence defined by certain scheduling statements. Such a statement will schedule an event normally to happen at some later time. An event is the execution of the next active phase of a specified process.

At the end of an active phase of a process the main control leaves that process to execute some other event. However, a reactivation point is defined for the process, telling where in the operation rule the next active phase shall start. Exceptions are the cases when control leaves the process through the final end of its operation rule or by a go to statement. In these cases no reactivation point is defined. However, the process may remain in the system as a data structure.

We notice that the local sequence control of a process is identical to the main control during active phases, and it is represented by the reactivation point during inactive periods.
2.3 States.

A process can be in one of four possible states, depending on whether a reactivation point is defined, and on whether an event has been scheduled for it and not completed. The states are defined in the following table.

<table>
<thead>
<tr>
<th>States</th>
<th>react. pt.</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>active</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>suspended</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>passive</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>terminated</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Immediately after the generation a process is passive with a reactivation point on the first statement of its operation rule. The evaluation of value parameters and subscript bounds of endogenous array attributes is part of the evaluation of the process designator.

2.4 Exogenous attributes.

The parameter (exogenous attribute) mechanism of an activity declaration is less general than that of a procedure declaration. All parameters, except arrays, are called by value. The call by value is implicitly understood, and value specification does not apply. Label, switch, and procedure parameters are not accepted.
The parameter mechanism is such that a process can not in general refer back to the block in which the corresponding process designator was evaluated. Therefore a process may remain and operate after the latter block is out of the system, i.e. the life spans of different processes may overlap each other in any way. A process may, however, refer directly to items non-local to itself, i.e. items local to the block containing the activity declaration (see CHAPTER 6), or to outer blocks.

An exogenous array attribute is assigned as a local name on the array designated by the actual parameter of the process designator. It follows that an array may have a life span exceeding that of the block in which it was declared.
CHAPTER 3.

ELEMENTS AND SETS.

3.1 Basic Concepts.

The element/set mechanism of SIMULA makes it possible to form and manipulate groups of objects in discrete event systems. The objects are processes.

The actual contents of a set are references to processes, rather than the processes themselves. An individual reference is called an element. A set is an ordered sequence of elements and can contain any number of elements. Its contents will change dynamically.

An element can be a member of at most one set. However, any number of elements can refer to a given process, which means that the process can "be" in any number of sets at a given time. There can be more than one element referring to the same process in one set.

Elements can be referenced dynamically through element expressions. This is a means, and the only means, of referencing individual processes: a process is always referenced indirectly by a dynamic reference to an element, i.e. by evaluating an element expression.

A process will remain in the system only as long as it is referenceable, which is (at most) as long as there is an element in the system referring to it.
The element/set concepts together with the "connection" mechanism described in CHAPTER 5 make SIMULA a very general list processing language. This is demonstrated in section 5.3.

3.2 Elements and Element Variables.

Elements are generated dynamically as the result of evaluating certain generative element expressions. The process reference of a given element remains fixed, but its set membership may change dynamically as the element goes in and out of sets or changes its position in a set.

An element of a set defines its own successor and predecessor in the set. It contains three references in all, as shown by fig. 1.

```
<table>
<thead>
<tr>
<th>reference to succeeding element</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference to preceding element</td>
</tr>
<tr>
<td>reference to process</td>
</tr>
</tbody>
</table>
```

Fig. 1

The references to the succeeding and preceding elements define the set membership, SM, of an element. The process referenced is often called its process aspect, PA.

An element which is currently not a member of any set is said to have no SM. The set membership references are both to "no element". Certain elements have no PA, i.e. the process reference is to "no process".
An element value is defined as either an individual element or "no element". The latter is denoted "none". none has no PA and no SM.

An element value is a value in the ALGOL sense of type "element". It is obtained by evaluating an element expression and can be assigned to an element variable in an assignment statement of the usual form.

< variable> := < element expression>

The assignment of a given element as the value of an element variable can alternatively be interpreted as the assignment of the variable as a name on the element. If the element has a PA the variable functions indirectly as a name on that process.

element variables can be simple or subscripted. Typical declarations are

    element X,Y,Z; element array crane [1:n];

The symbol "element" is a type declarator in every respect similar to the declarators real, integer, and Boolean of ALGOL.

3.3 Sets.

A set is a cyclic sequence of elements, of which all except one have process aspects. The one with no PA is called the set head. It is always present and is always the same element for a given set.
The set head is to be regarded as a dummy element, it functions as the "end" of the set in both directions. An empty set is represented by a sethead which is its own successor and predecessor. In a non-empty set the successor of the set head is called the first element, and the predecessor of the set head is called the last element of the set.

A set has fixed name, called a set designator, which can be simple or subscripted. Set declarations are syntactically similar to those of simple and subscripted variables.

```
set S,T,U; set array file[1:n];
```

However, a set designator is not a variable in the sense that "set values" may be assigned explicitly. There is no "set expression" concept in the language, and thus no "set procedure". Although the contents of a set may differ from time to time, a set designator denotes the same set throughout its scope. The set head is generated as a result of the declaration and stays in the set all the time. A set is empty initially.
The call by value of a set parameter has the significance of assigning the formal parameter as a local name on the set denoted by the actual parameter. If the parameter is an exogenous attribute of a process, it will be seen that the set may remain in the system longer than the block in which it was declared.

A set ceases to exist when it looses its (last) name, because of exit from a block or because a process leaves the system. Then all its elements loose their set membership including the set head. The latter becomes a **void** element having no PA and no SM. The elements, or some of them, may remain in the system. See also section 10.3.

3.4 **Element Expressions**.

As previously stated elements are generated as the result of evaluating certain **element** expressions, called **generative** expressions, or as the dynamic result of a set declaration. A generated element will stay in the system as long as it is **referenceable**.

A reference to an element can be through

1. an **element** variable, or
2. set membership, or
3. the sequencing set (see CHAPTER 4), or
4. connection (see CHAPTER 5).

There are **element** expressions corresponding to each of these cases.

Most **element** expressions are function designators referencing **element** procedures. The only exceptions are the **element** "constant" **none**, **element** variables (and value parameters), process designators, and activity identifiers in certain contexts.
Most element procedures are expressible within the SIMULA language in terms of basic procedures expressed in machine code. For convenience also a number of non-basic element procedures are expressed in machine code, and may be considered part of the language. The same is true for procedures of other kinds dealing with elements and sets.

Throughout the remainder of this report the letters X,Y,Z are used to represent element expressions. The letter S usually denotes a set.

3.4.1. Generative expressions.

The value of a generative expression is a new element, i.e. an element whose identity differs from every other element currently present in the system. SIMULA has two generative expressions:

1. Process designator.

   A process designator has one of the forms
   A(<actual parameter list>) or new A(<actual parameter list>)

   where A is an activity identifier. If the activity A has no
   parameters the process designator is simply
   A or new A.

   The symbol "new" only serves to resolve an ambiguity of the
   identifier A inside the activity declaration A itself (see 4.4)
   and inside a connection block connecting a class A process
   (see CHAPTER 5), in which cases the construction "new A" must
   be used. Elsewhere the symbol new is redundant, but can often
   be used to advantage to improve readability.

   The element value of a process designator has a PA, but no
   SM. It refers to the process generated as a result of the
   evaluation.
Examples.

a. `new clerk(true, false, 10),`

where

```
activity clerk (redheaded, greeneyed, thumbs);
Boolean redheaded, greeneyed; integer thumbs;
begin ------------------------------- end;
```

b. `X := Y := new A; Z := new A;`

Now X and Y denote the same element; Z denotes another one. The two elements refer to different processes.

2. `proc(X).`

`proc(X)` is a new element with the same process aspect as X, and no SM. If X has no PA, the value is `none`.

Examples.

a. `X := new A; Y := proc(X); Z := proc(X);`

Now X, Y and Z denote different elements, which all refer to the same process.

b. `X := proc(new A);`

The element generated as the result of evaluating the process designator can not be referenced and therefore will leave the system. The element referenced by X remains, and so does the generated process.

3.4.2 Set membership references.

1. `head(S)` denotes the set head of the set S. It has a SM and no PA, and functions as a dummy element always present in the set.
2. \text{suc}(X) \text{ denotes the successor of } X. \text{ If } X \text{ has no SM,} \\
\text{suc}(X) \text{ is } \text{none}. \text{ If } X \text{ has a SM,} \text{suc}(X) \text{ is a unique element} \\
having itself a SM.

3. \text{pred}(X) \text{ denotes the predecessor of } X. \text{ If } X \text{ has no SM,} \\
\text{pred}(X) \text{ is } \text{none}. \text{ If } X \text{ has a SM pred}(X) \text{ is a unique} \\
element having itself a SM.

\text{suc} \text{ and } \text{pred} \text{ are reciprocal functions. It is more a convenience} \\
than a necessity to have both as basic procedures.

3.5 \textbf{Boolean expressions.}

The following are basic \textbf{Boolean} expressions applying to elements.

1. "\text{X} = \text{Y}" is \textbf{true} if either \text{X} and \text{Y} denote the same element \\
or both values are \textbf{none}.

2. "\text{X} \neq \text{Y}" is the negation of "\text{X} = \text{Y}".

3. \text{same}(X,Y) \text{ is } \textbf{true} if either \text{X} and \text{Y} reference the same \\
process or neither has a PA. \\
"\text{X} = \text{Y}" implies \text{same}(X,Y).

4. \text{similar}(X,Y) \text{ is } \textbf{true} if either \text{X} and \text{Y} reference processes \\
of the same class or neither has a PA. \\
\text{same}(X,Y) \text{ implies } \text{similar}(X,Y).

\textbf{Examples.}

"\text{X} \neq \text{X}" is \textbf{true} if \text{X} is a generative expression. \\
\text{same}(X, \text{proc}(X)) \text{ is } \textbf{true} if \text{X} evaluates to the same element both \\
times. \\
\text{similar}(\text{new A}, \text{new A}) \text{ is } \textbf{true}. \\
\text{same}(\text{head}(S), \text{none}) \text{ is } \textbf{true}, \text{ but } "\text{head}(S) = \text{none}" \text{ is } \textbf{false}. 
3.6 Element Operations.

The following statements are the basic operations available for manipulating sets. The procedures operate on the SM of elements without changing their identities.

1. pred(X,Y)
   If X has a SM, say, pred(X) = Z, Z ≠ none, and Y has a PA and no SM, then suc(Y) becomes X and pred(Y) becomes Z, and the SM of X and Z are modified so that pred(X) and suc(Z) both become Y. Otherwise the statement has no effect. In the case quoted Y is given a SM, intuitively by inclusion in a set between X and its predecessor.

Example.

The statement pred(head(S), new A) will include the generated element as the last one of S. The previous last one becomes next to the last.

2. remove(Y)
   If Y has a SM and a PA, say, X = suc(Y) and Z = pred(Y), X,Z ≠ none, then suc(Y) and pred(Y) become none, suc(Y) becomes X, and pred(X) becomes Z. Otherwise the statement has no effect.
   In the case quoted Y looses its SM or, intuitively, Y is removed from the set of which it was a member, and the former predecessor and successor of Y become consecutive elements.
The fact that neither of the above statements has an effect if Y has no PA, shows that only elements referring to processes can go in and out of sets. A set head must remain in its set, and there can be only one such element in the set.

We emphasize once more that the statements operate on properties of elements, without changing the identities of these elements.

Example.

\[ \text{pred}(Z,X) \ ; \ Y := X \ ; \ \text{remove}(X); \]

Now Y has no SM. Reason: X and Y have the same element value, i.e., they denote the same element.

3.7 Non-Elementary Procedures.

The following procedures are available as machine code procedures, although they are all expressible in terms of basic SIMULA concepts.

**element procedures**

1. **first(S)** is equivalent to \(\text{suc}(\text{head}(S))\).
2. **last(S)** is equivalent to \(\text{pred}(\text{head}(S))\).
3. **successor(n,X)** is intuitively defined as \(\text{suc}^n(X)\), i.e., by stepping \(\text{abs}(n)\) places forward or backward according to the sign of \(n\). The stepping is discontinued if and when the set head is reached.

```simula
  element procedure successor(n,X); value n, X;
  integer n; element X;
  begin integer i;
```
\[ i := 0; \]
\[ \text{for } i := i+1 \text{ while } i \leq \text{abs}(n) \land \neg \text{same}(X, \text{none}) \text{ do} \]
\[ X := \text{if } n > 0 \text{ then suc}(X) \text{ else pred}(X); \]
\[ \text{successor} := X; \text{ end}; \]

4. \text{number}(n,S) \text{ is equivalent to successor}(n, \text{head}(S)). \text{ This function defines a consecutive double numbering of the elements of a set, such that \text{first}(S) is "number 1" and \text{last}(S) is "number -1".}

5. \text{member}(X,S) \text{ denotes an element of } S \text{ with the same PA as that of } X. \text{ If there is no such element the value is none. If there is more than one, the element is taken which has the smallest positive ordinal number.}

\begin{verbatim}
procedure member(X,S); value X,S;
element X; set S;
beg element Y; member := none;
for Y := head(S), suc(Y) while Y \neq head(S) do
  if same(X,Y) then begin member := Y; go to fin end;
fin:
end;
\end{verbatim}

\textbf{Boolean procedures}

1. \text{exist}(X) \text{ is equivalent to } \neg \text{same}(X, \text{none}); i.e. the value is true if \text{X} has a process aspect.

2. \text{empty}(S) \text{ is equivalent to } \neg \text{exist(first}(S))

\textbf{integer procedures}

1. \text{ordinal}(X) \text{ denotes the positive ordinal number of } X, \text{ if } X \text{ has a PA and a SM. Otherwise the value is zero.}

\begin{verbatim}
procedure ordinal(X); value X; element X;
beg integer i; i := 0; X := suc(X);
for X := pred(X) while exist(X) do i := i+1;
ordinal := i
end;
\end{verbatim}
2. cardinal(S) denotes the number of (non-trivial) elements of S. It is equal to ordinal(last(S)).

Statements

1. precede(X,Y) is equivalent to prcd(X,Y), except that Y is first removed from the set of which it was a member, if any.

   procedure precede(X,Y); value Y; element X,Y;
   begin remove(Y); prcd(X,Y) end;

2. follow(X,Y) is equivalent to precede(suc(X),Y).

3. transfer(X,S) is equivalent to precede(head(S),X). X is removed from its set, if any, and included as the last element of S.

4. include(X,S) will include a new element with the same PA as that of X, or X itself, as the last element of S, depending on whether or not X has a SM already.

   procedure include(X,S); value X,S;
   element X; set S;
   prcd (head(S), if suc(X) = none then X else proc(X));

5. clear(S) will remove all elements of S, making S an empty set.

   procedure clear(S); value S; set S;
   begin element X;
   for X: = first(S) while exist(X) do remove(X) end;

3.8 Examples.

1. The verdict.

   set herd, barn, stable; element X, sheep;
   -------------------------------
   for X: = first(herd) while exist(X) do
   transfer (X, if similar (X, sheep) then barn else stable);
Notice that the elements of the herd set are removed by the transfer statement, so that the expression first(herd) evaluates to another element each time.

2. **Set subtraction.**

Those elements of S are removed that represent processes also in T.

```plaintext
set S, T; element X;

X := head(S);
for X := suc(X) while exist(X) do
  if exist(member(X, T)) then begin X := pred(X); remove (suc(X)) end;
```

Notice that the remove procedure will terminate the scan loop if applied to X itself, since after removal of X suc(X) is `none`.

The following is a simpler way to achieve the same thing, which will work provided that there is at most one reference in S to each process in T.

```plaintext
for X := first(T), suc(X) while exist(X) do remove(member(X, S));
```
4.1 The Sequencing Set.

An event is an active phase of a process. When an event is scheduled, an event notice is generated referring to the process to become active. The event notice is included in the sequencing set, SQS, which is an ordered set of event notices. There can be at most one event notice referencing a given process.

The reference from an event notice to a process is through an element. I.e. the event notice refers to an element, which in turn references the process. The referenced element may or may not be a member of a set; its own SM has nothing to do with the SQS.

The SQS contains all event notices referring to events scheduled but not yet completed. Its first member is the event notice corresponding to the event currently in execution, the current event. The other events referred to in the SQS are called future events. The event whose event notice is the successor of the event notice of the current event is called the next event.

When an event is completed, its event notice is removed from the SQS and the next event becomes the current event.

The currently active process is referenced by the element expression "current". The element is the one referred to by the current event notice.
Statements operating on the SQS are called sequencing statements. The execution of a sequencing statement may terminate the current event. An event is terminated either by executing a sequencing statement, or by leaving the active process, through the final end of its operation rule, or by going to a label not local to the process.

A sequencing statement generating an event notice is called a scheduling statement. A scheduling statement assigns a real number to the event notice, which determines its position in the SQS. Because of its use in system descriptions the number is called the time reference of the event notice. The time reference of an individual event notice remains fixed.

The time reference of the current event is called the current system time. It is accessed by the real procedure "time". The SQS can be regarded as a representation of the "system time axis". The fact that the system time remains constant during an active phase of a process, shows that the execution of a SIMULA program is a sequence of discrete events, when viewed in system time.

A scheduling statement will insert the generated event notice in the SQS behind all event notices with a smaller time reference and in front of all events with a greater time reference. The new event notice normally goes in behind those with the same time reference, if any; but the scheduling statement can specify insertion with priority, in which case the event notice is placed in front of those with the same time reference. Its position can also be explicitly stated.

If an event notice with the time reference equal to the current system time is inserted with priority, it takes precedence over the current event notice. The current event becomes the next event (a reactivation point is defined for the process), and the scheduled event becomes the current event. The effect of such a scheduling statement is to some extent similar to that of a procedure call.
4.2 Sequencing Statements.

The following sequencing statements may operate on the SQS by removing an event notice. If it is the current event notice, the current active phase is terminated. The state of the referenced process is changed accordingly. The element through which the process is referenced need not be the same as the one referred to by the event notice.

1. terminate(X).

The process referred to by X becomes terminated. Any event notice referring to the process is removed from the SQS, and any reactivation point is deleted. The effect is as if control had left the process through its final end. If X has no process aspect the statement has no effect.

With the exception of this statement any sequencing statement, including those discussed in the following section, will define the reactivation point of the currently active process, if the current active phase is terminated. The reactivation point is on the next statement in the dynamic sense.

2. cancel(X).

If the process referred to by X is active or suspended, the associated event notice is removed from the SQS, and the process becomes passive. A reactivation point is defined if the process was active, and is left unchanged if it was suspended. In all other cases the statement has no effect.

3. passivate.

This statement is equivalent to cancel(current).

4. wait(S).

This statement is equivalent to the compound statement begin include(current, S); passivate end.
4.3 Scheduling Statements.

The following sequencing statements may schedule an event, and some may in addition cancel an event already in the SQS.

The statement

\[ \text{hold}(T) \]

where \( T \) is an arithmetic expression with a non-negative value, will cancel the current event, terminating the current active phase, and schedule a new event for the same process. The event notice gets the time reference "time + T", and is included in the SQS without priority. Viewed in system time the statement normally represents a suspension period of length \( T \) for the current process, after which control proceeds to the next statement in the dynamic sense. Notice, however, that the event scheduled for the current process may be cancelled by another process before it is executed.

Since the future event is inserted without priority, the statement \( \text{hold}(0) \) will allow further events with the same time reference as the current one to be executed, after which the current process resumes control. If \( T \) has a negative value, \( \text{hold}(T) \) is equivalent to \( \text{hold}(0) \).

With the exception of the "hold" statement all scheduling statements follow a special syntax which allows greater flexibility than do ordinary procedure statements. The general format is

\[ \langle \text{activation clause} \rangle \langle \text{scheduling clause} \rangle \]

in which the latter clause is optional. Connection clauses may be added (see CHAPTER 5).
A typical scheduling statement is

activate X at T,

where X is an element expression and T is an arithmetic expression. The statement has an effect if and only if X refers to a passive process. Viewed in system time the next active phase of that process is scheduled for the time T.

An event notice is generated with the time reference equal to the greater of T and the current system time. It is inserted in the SQS without priority. The event notice refers to the element which is the current value of the expression X. This element becomes the value of the function "current" during the execution of the scheduled active phase. Notice that the SM of the element (but not the PA may have been altered in the meantime).

Another option is to write

activate X delay T;

where the time reference is specified relative to the current system time. The scheduling clause "delay T" is equivalent to "at time + T".

Scheduling with priority is by writing

activate X at T prior; or activate X delay T prior;

The statement

activate X delay 0 prior; (or at time prior)

has the equivalent abbreviated form

activate X;
The event is scheduled in front of the current event, so that the indicated active phase is executed immediately in real time. The current active phase is terminated, but its event notice will remain in the SQS as the next event. Provided that this event is not cancelled, the indicated active phase functions as a "subroutine" to the current process. This is called direct scheduling.

The statements

activate X after Y ; and activate X before Y ;

have an effect only if the process referenced by Y is active or suspended, i.e. if there is an event notice referencing the process (not necessarily through the same element). The event notice generated for X is included in the SQS immediately after, or immediately before, the specified event notice. It is given the same time reference as the latter. The priority concept does not apply.

activate X before current;

is another way of specifying direct scheduling.

No activate statement has an effect if the process designated by X is active or suspended, i.e. if it is already represented in the SQS. However, similar statements with "activate" replaced by "reactivate" will have an effect in this case as well. The old event notice, if any, referring to the designated process is removed from the SQS, and then a new event is (or may be) scheduled as above.

If the designated process is not the currently active one, the statement
reactivate X at T;
has the same effect as

cancel(X); activate X at T;
(provided that the expression X evaluates to the same element both times and has no side effect).

The statement

reactivate current delay T;
is equivalent to hold(T).

If X refers to the currently active process (same(X, current) is true), the statement

reactivate X;
has no other visible effect than possibly changing the element identity (and set membership aspect) of "current".

The statement

reactivate X before Y; (or after Y)
has the same effect as cancel(X), if Y does not refer to an active or suspended process.

No scheduling statement will have an effect if the element X has no process aspect, or if the designated process is terminated.

4.4 SQS Functions.
The following function procedures are based on the SQS concept.
Element procedures.

1. current.

   The value is the element referred to by the current event notice. This element references the currently active process, i.e. the process in which the expression is evaluated.

2. Activity identifier.

   An activity identifier referenced within the corresponding activity declaration is a function designator equivalent to "current" except when preceded by the symbol "new" (see section 3.4.1) or the symbol "when" (see CHAPTER 5). When referenced within the body of a procedure declared local to the activity, it may occasionally assume a different significance (see CHAPTER 5).

3. nextev(X).

   If the PA of X is scheduled as an event in the SQS, i.e. is an active or suspended process, the function value refers to the process scheduled as the following event. The element is the one referenced by the corresponding event notice. The value is none if there is no following event in the SQS, or if the PA of X is passive or terminated, or if X has no PA.

Real procedures.

1. evtime(X).

   The value is equal to the time reference of the event notice corresponding to the referenced process. If the process is passive or terminated, or if X has no PA, the value is undefined.

2. time.

   The value is equal to evtime(current).
Boolean procedures.

1. idle(X).
   The value is true if X refers to a passive or terminated process, or if X has no PA.

2. finished(X).
   The value is true if X refers to a terminated process, or if X has no PA.

Examples.
The state of a process can be assessed by the following Boolean expressions:

active: same(X, current),
suspended: \neg idle(X) \wedge (\neg same(X, current)),
passive: idle(X) \wedge (\neg finished(X)), and
terminated: finished(X)

4.5 Examples.

1. Postponement.

   reactivate X at evtime(X) + T;

   Provided that the referenced process is suspended
   the statement will increase its suspension period by T.

2. Scheduling for an unknown system time.
   If X refers to a suspended process the statement

   reactivate current after X;

   will invoke a suspension period of the currently active process
   such that its next active phase comes immediately after that of
   X (provided that neither of these events is cancelled before
   execution).

   A similar effect can be obtained by means of the following
procedure, if the PA of $X$ may be a passive process. It is assumed that $X$ is an element variable, and that all relevant activate statements in the system refer to the process via this variable.

\[
\text{procedure } \text{link}(Z); \ \text{element } Z; \\
\text{if } \text{idle}(Z) \ \text{then } \begin{align*}
\text{begin } & \text{element } \text{swap;} \\
& \text{swap} : = Z; \ Z : = \text{current}; \ \text{passivate}; \\
& \text{Z} : = \text{swap}; \ \text{activate } Z \ \text{end} \\
\text{else } & \text{reactivate current after } Z;
\end{align*}
\]

Notice that the element "current" refers to the process containing the call for the link procedure currently in execution. The passivate statement also pertains to that process. It follows that the statement

\[
\text{link}(X)
\]

invokes an inactive period (passive or suspended) of the currently active process of the required length.

3. Debugging.

A process of the following class will "screen" the events executed in the system, except events invoked by direct scheduling. Special actions can be taken before and after active phases of the process specified by the exogenous element attribute $X$, e.g. debugging information can be printed.

\[
\text{activity monitor}(X); \ \text{element } X; \\
\text{begin element } Y; \\
\text{next: } Y : = \text{nextev(current)}; \\
\text{rep: if } \text{similar}(Y, \text{current}) \ \text{then} \\
\begin{align*}
& \text{begin } Y : = \text{nextev}(Y); \ \text{go to rep end;}
\end{align*} \\
& \text{if } \text{same } (X,Y) \ \text{then} \\
\begin{align*}
& \text{begin } \text{reactivate current before } Y;
\end{align*}
\]
actions1; reactivate Y; actions2 end
else reactivate current after Y;
go to next end;

The short loop beginning at label "rep" makes it possible for any number of monitor processes to operate in parallel. Any number of processes can therefore be monitored.

The monitor processes will not interfere with the operation of the system, provided that there is no reference to the procedure "nextev".
CHAPTER 5.

CONNECTION

5.1 Connection Statements.

The connection mechanism of SIMULA provides a means of interaction between processes. A process can, by connecting another one, get access to the attributes of the latter.

A connection statement has the following general form.

\[
\text{inspect } X \text{ when } A_1 \text{ do } S_1 \\
\text{when } A_2 \text{ do } S_2 \\
\vdots \\
\vdots \\
\text{when } A_n \text{ do } S_n \\
\text{otherwise } S;
\]

where \( A_1, A_2, \ldots, A_n \) are activity identifiers, and \( S_1, S_2, \ldots, S_n, \) and \( S \) are statements. A construction "when \( A_i \text{ do } S_i \)" is called a connection clause. There can be any number of them and must be at least one. The otherwise clause is optional.

If the PA of \( X \) is a process belonging to class \( A_1 \), the statement \( S_1 \) is executed, and the other statements are ignored. The connection is said to be effective during the execution of \( S_1 \). The value of the element expression \( X \) becomes the connected element, and the referenced process becomes the connected process. If the process does not belong to any of the classes listed, or if \( X \) has no PA, the statements \( S_1, \ldots, S_n \) are skipped and the statement \( S \) is executed, if present.
Each statement $S_i$ is a connection block. It is interpreted as if it were a part of the outermost block of the activity $A_i$, in the sense that the exogenous and endogenous attributes of the connected process are immediately accessible through their local names. The block containing the connection statement acts as an outer block. Another connection statement is

$$\text{extract } X \text{ when } \ldots \text{ etc}$$

It has the same general form as the one above, and also the same effect, except that the element $X$ is removed from its set, if it has a SM and a PA.

If the expression $X$ is a process designator, the class of the process is known. In this case the alternative connection clauses can be replaced by the single construction "do $S$", where $S$ is a statement serving as a connection block. No otherwise clause applies in this case.

activate and reactivate statements can be augmented by connection clauses. If the scheduling is direct, the indicated active phase is executed before connection becomes effective, otherwise after, possibly while connection is effective. Connection may become effective even if no event is scheduled.

Within a connection block connecting a class A process the activity identifier $A$, not preceded by the symbol "new" or by the symbol "when", has the significance of a function designator referencing the connected element. The same is true for a reference to $A$ within the body of a procedure declared local to $A$, when the procedure is called within the connection block.
Example.

```
activate new car do if speed > 50 then include(car, left lane)
else include(car, right lane);
```

where "speed" is an endogenous attribute of the car process,
whose value is defined during the first active phase of the
process.

Because of its use in connection statements an activity identi-
 fier may not be represented by a formal parameter. Activity
 identifiers are under no circumstances permitted as parameters
to procedures or activities. There is no "activity" specifier
in the language.

5.2 Label Attributes.

The fact that labels, switches and procedures local to the
outermost block of the specified activity body are accessible
within a connection block, makes it possible to "enter" a
connected process. This can happen as the result of an ex-
plicit go to statement within the connection block, or as a
side-effect of a procedure local to the activity called within
the connection block.

Let L be a label local to the connected process. Then "go to L"
has the following effects:

1. The connected process is terminated, i.e. any event notice
   referring to it is removed from the SQS, any reactivation
   point for the process is deleted.

2. The current process is terminated without removal of the
   current event notice. Connection is thereby cancelled.
3. The element reference of the current event notice is replaced by a reference to the formerly connected element, which thereby becomes "current".

4. An active phase of the formerly connected process commences at the specified label L. The system time remains unchanged.

By this means the reactivation point of a currently passive or suspended process can be superseded and a terminated process can be "revived".

5.3 Examples.

1. Procedure Attributes.

Cars are traveling on a road. Each car is characterized by its velocity V and its position X. The former is a step function and the latter a continuous function of time.

```
activity car;
begin real V, Xo, To;
   real procedure X; X := Xo + VX(time - To);
   procedure update (Vnew); real Vnew;
   begin Xo := X; To := time; V := Vnew end;
```

end;

To is the time when V was last updated, and Xo is the position of the car at that time.

A regular police survey tries to enforce an upper speed limit Vmax on a bad portion of the road, between X1 and X2.
set road, police file; real X1, X2, Vmax;

activity survey (interval); real interval;
begin element Z;
scan: for Z := first(road), suc(Z) while exist(Z) do
    inspect Z when car do
        if Z < X1 ∧ Z < X2 ∧ V > Vmax then
            begin update(Vmax); include(car, police file) end;
            hold(interval); go to scan
end;

Notice that the procedures "X" and "update" referenced within the connection block are those declared for the currently connected car. The non-local items referenced within the bodies of these procedures are therefore the attributes of this particular car.

2. List Processing.

Let T be a set, some of whose elements are "branch" processes.

set T;
activity branch; begin set subtree;----- end;

T can be regarded as a tree structure if elements of a subtree may be branch processes. The "leaves" of the tree can be defined as those elements of T or of a subtree which are not branch processes.

A tree structure like this can be scanned "leafwise" by means of a recursive procedure. The following activity is equivalent to a "reader" concept found in well known list processing languages. It incorporates a recursive scan procedure,

activity leafscan (tree); set tree;
begin element leaf;
    procedure scan(S); set S;
    begin element X; X := head(S);
        for X := suc(X) while exist(X) do


inspect X when branch do scan(subtree)
otherwise begin leaf: = X; passivate end
end scan;
scan(tree); leaf: = none
end leafscan;

A reader on T can be initialized by the following statements.

Tsc := new leafscan(T); activate Tsc;

where Tsc is an element variable. Later activate statements on Tsc will step the pointer, which is the endogenous attribute "leaf" of the leafscan process. When the elements are exhausted, leaf becomes none.

Access to the pointer is by connecting the element Tsc, explicitly or through one of the following procedures:

element procedure reader(Z); element Z;
inspect Z when leafscan do reader := leaf;
element procedure next(Z); element Z;
activate Z when leafscan do next := leaf;

The expressions reader(Tsc) and next(Tsc) both evaluate to the element of the tree T presently under observation. The latter has the side effect of stepping the pointer.

Notice that the pointer is stepped before the element value is assigned, since the activation is by direct scheduling.

In general list structures of processes can be formed by means of set or element attributes. The user has complete freedom when defining the structures of the various components of a list.
The list processing facilities of SIMULA can be exploited for their own sake with no reference to the discrete event system concept. In such cases it may be natural to define the processes on a list as passive data carriers.

3. Sorting.

Given a set, file1, containing references to "record" processes (and possibly processes of other classes). We want to establish another set, file2, of references to these record processes, sorted against a real attribute "key" with nonnegative values.

```
set file1, file2;
activity record; begin real key; --- end;
```

The following piece of program performs the sorting by means of the SQS. It is assumed that all record processes in the system are passive.

file2 is empty initially.

```
begin element X,Y;
for X := first(file1), suc(X) while exist(X) do
    inspect X when record do activate X delay key;
    X := current; Y := none;
for X := nextev(X) while exist(X) do
    inspect X when record do
        begin cancel(Y); Y := X; include(X, file2) end;
    cancel(Y) end;
```

The latter inspection clause only serves to skip processes referenced in the SQS not belonging to the "record" activity.
CHAPTER 6.

THE MAIN PROGRAM

A discrete event system description is written formally as a SIMULA block.

```
SIMULA begin -------------- end
```

A SIMULA block is itself a statement, which may be a part of an otherwise ordinary ALGOL program. The SIMULA concepts introduced in the preceding chapters are only available within a SIMULA block. A program may contain more than one SIMULA block; however, nested SIMULA blocks are not permitted.

The SQS of a given system is local to the SIMULA block. The SIMULA block head must contain all activity declarations in the system. Activity declarations are not permitted elsewhere. Element and set declarations can be given anywhere inside a SIMULA block, i.e. also within sub-blocks of any kind, including procedures and activities.

The SIMULA block functions dynamically as a process, which will be called the main program. It is always present in the system. The initial system setup is conveniently done in the main program. All sequencing statements have their usual significance.

Upon entry into a SIMULA block the main program is the currently active process, and the current system time is equal to zero. The SQS contains one event notice referencing an element with no SM, which in turn refers to the main program.
The apparent paradox that the main program should function both as a process in parallel with other processes and as an outer block to all processes (therefore to itself), is resolved by defining the outer block to a process to be a connection block connecting the main program (i.e. the SIMULA block). The fact that the latter has itself connected in an immediately outside block causes no difficulty. The connection is through the element mentioned above.

A go to statement leading from within the body of an activity, i.e. from a process, into the main program has the usual effect of a go to statement leading into a connected process (see section 5.2).

When leaving the SIMULA block through its final end the simulation is terminated, even if further events have been scheduled. The same is true if a go to statement is executed, leading out of the SIMULA block.

If the SQS becomes empty before exit from the SIMULA block is made, further actions are undefined. This is possible only if all processes, including the main program, are passive or terminated.

The statement terminate(X), where X refers to the main program, will not terminate the simulation. Normal termination can still be achieved by executing a go to statement leading to a suitable point in the main program or leading out of the SIMULA block.
CHAPTER 7.

RANDOM DRAWING.

7.1 Pseudo-random Number Streams.

All random drawing procedures of SIMULA use the same technique of obtaining basic drawings from the uniform distribution in the interval \(<0,1>\).

A basic drawing will replace the value of a specified integer variable say, \(U_i\), by a new value according to the following algorithm.

\[ U_{i+1} = \text{remainder} \left( \left( U_i \times 5^{2p+1} \right) \times 2^n \right), \]

where \(U_i\) is the \(i\)'th value of \(U\).

It can be proved that, if \(U_0\) is a positive odd integer, the same is true for all \(U_i\); and the sequence \(U_0, U_1, U_2, \ldots\) is cyclic with the period \(2^{n-2}\). (The last two bits of \(U\) remain constant, while the other \(n-2\) take on all possible combinations.) In UNIVAC 1107 we have \(n = 35\). \(p\) is chosen equal to 6.

The real numbers \(u_i = U_i \times 2^{-n}\) are fractions in the range \(<0,1>\). The sequence \(u_1, u_2, \ldots\) is called a stream of pseudo-random numbers, and \(u_i\) (\(i = 1, 2, \ldots\)) is the result of the \(i\)'th basic drawing in the stream \(U\). A stream is completely determined by the initial value \(U_0\) of the corresponding integer variable. Nevertheless it is a "good approximation" to a sequence of truly random drawings.
by reversing the sign of the initial value $U_0$ of a stream
variable the antithetic drawings $1 - u_1, 1 - u_2, \ldots \ldots$ are
obtained. In certain situations it can be proved that means
obtained from samples based on antithetic drawings have a
smaller variance than those obtained from uncorrelated streams.
This can be used to reduce the sample size required to obtain
reliable estimates.

7.2 Random Drawing Procedures.

The following procedures all perform a random drawing of some
kind. Unless otherwise is explicitly stated the drawing is
effected by means of one single basic drawing, i.e. the
procedure has the side effect of advancing the specified
stream by one step. The necessary type conversions are effected
for the actual parameters, with the exception of the last one.
The latter must always be an integer variable specifying a
pseudo-random number stream. All parameters except the last
one are called by value.

1. **Boolean procedure** draw (a, U); real a; integer U;

   The value is true with the probability $a$, false with the
   probability $1 - a$. It is always true if $a \geq 1$, always
   false if $a \leq 0$.

2. **integer procedure** randint (a, b, U); integer a, b, U;

   The value is one of the integers $a, a + 1, \ldots, b - 1, b$
   with equal probability. It is assumed that $b \geq a$.

3. **real procedure** uniform (a, b, U); real a, b; integer U;

   The value is uniformly distributed in the interval
   $[a, b]$. It is assumed that $b > a$. 

4. real procedure normal (a, b, U); real a, b; integer U;

The value is normally distributed with mean a and standard deviation b. An approximation formula is used for the normal distribution function:


5. real procedure psnorm (a, b, c, U); real a, b; integer c, U;

The value is formed as the sum of c basic drawings, suitably transformed so as to approximate a drawing from the normal distribution. The following formula is used:

\[ a + b \left( (\sum_{i=1}^{c} u_i) - c/2 \right) \sqrt{12/c} \]

This procedure is faster, but less accurate than the preceding one. c is assumed ≤12.

6. real procedure negexp (a, U); real a; integer U;

The value is a drawing from the negative exponential distribution with mean 1/a, defined by -ln(u)/a, where u is a basic drawing. This is the same as a random "waiting time" in a Poisson distributed arrival pattern with expected number of arrivals per time unit equal to a.

7. integer procedure Poisson (a, U); real a; integer U;

The value is a drawing from the Poisson distribution with parameter a. It is obtained by n+1 basic drawings, u_i, where n is the function value. n is defined as the smallest non-negative integer for which

\[ \prod_{i=0}^{n} u_i < e^{-a}. \]
The validity of the formula follows from the equivalent condition
\[ \sum_{i=0}^{n} -\ln(u_i)/a > 1, \]
where the left hand side is seen to be a sum of "waiting times" drawn from the corresponding negative exponential distribution.

When the parameter \( a \) is greater than 20.0, the value is approximated by integer (normal \((a, \sqrt{a}), u\)) or, when this is negative, by zero.

8. **real procedure** Erlang \((a, b, U); \ value a, b; \ real a, b; \ integer U; \)

The value is a drawing from the Erlang distribution with mean \( 1/a \) and standard deviation \( 1/(a \sqrt{b}) \). It is defined by \( b \) basic drawings \( u_i \), if \( b \) is an integer value,

\[-\sum_{i=1}^{b} \frac{\ln(u_i)}{a \cdot b},\]

and by \( c+1 \) basic drawings \( u_i \) otherwise, where \( c \) is equal to \( \text{entier}(b) \),

\[-\left(\sum_{i=1}^{c} \frac{\ln(u_i)}{a \cdot b}\right) - \frac{(b-c) \cdot \ln(u_{c+1})}{a \cdot b}\]

Both \( a \) and \( b \) must be greater than zero.
9. integer procedure discrete (A, U); array A; integer U;
The one-dimensional array A, augmented by the element 1 to the
right, is interpreted as a step function of the subscript, de-
fining a discrete (cumulative) distribution function. The array
is assumed to be of type real.

The function value is an integer in the range \([\text{lsb}, \text{usb}+1]\), where
\text{lsb} and \text{usb} are the lower and upper subscript bounds of the array.
It is defined as the smallest \(i\) such that \(A[i] > u\), where \(u\) is a
basic drawing and \(A[\text{usb}+1] = 1\).

10. real procedure linear (A, B, U); array A, B; integer U;
The value is a drawing from a (cumulative) distribution function
\(F\), which is obtained by linear interpolation in a non-equidistant
table defined by \(A\) and \(B\), such that \(A[1] = F(B[1])\).

It is assumed that \(A\) and \(B\) are one-dimensional real arrays of the
same length, that the first and last elements of \(A\) are equal to
0 and 1 respectively and that \(A[i] > A[j]\) and \(B[i] > B[j]\)
for \(i > j\).

11. integer procedure histd (A, U); array A; integer U;
The value is an integer in the range \([\text{lsb}, \text{usb}]\), where \text{lsb} and
usb are the lower and upper subscript bounds of the one-dimen-
sional array \(A\). The latter is interpreted as a histogram defining
the relative frequencies of the values.

This procedure is more time-consuming than the procedure discrete,
where the cumulative distribution function is given, but it is
more useful if the frequency histogram is updated at run time.
CHAPTER 8.

DATA ANALYSIS

1. procedure histo (A, B, c, d); array A, B; real c, d;
will update a histogram defined by the one-dimensional
arrays A and B according to the observation c with the weight d.
A [i] is increased by d, where i is the smallest integer such
that c ≤ B [i]. It is assumed that the length of A is one
greater than that of B. The last element of A corresponds
to those observations which are greater than all elements
of B. The procedure will accept parameters of any combi-
nation of real and integer types.

2. procedure accum (a, b, c, d); value d; real a, b;
begin a := a + c x (time-b);
   b := time; c := c + d end;

This procedure serves to accumulate the system time integral
a = ∫ c dt. The parameters a, b, c are called by name. The
corresponding actual parameters must be variables. a and b
must be of type real, c and d can be real or integer.

Statements of the form

accum(ax,tx,x,dx)

will update the variable x by adding dx, as well as the
integral ax. dx can be any arithmetic expression. The variable tx serves to record the system time at which ax and x were last updated. During the period of integration no updating of ax, tx, and x should take place, except through the procedure. The initial value of tx should be the system time at which the integration is started, say ti. The variable ax can be updated to the current value of the integral at any time by the statement

\[ \text{accum(ax,tx,x,0).} \]

The average value of x is equal to \( ax/(\text{time}-ti) \).

Since the procedure refers to the current system time, it will only be available within a SIMULA block.

3. **procedure** hprint \((Y,X,\text{LB},UB,Y0,YS)\);

\( Y \) and \( X \) are arrays of type **integer** or **real**. \( \text{LB},\text{UB},Y0,YS \) are simple variables of type **integer** or **real**.

A call to this procedure will give a printout of a histogram. The columns are printed as 3 lines of asterisks (*).

The parameters have the following meaning:

- **Y**: Column length
- **X**: Upper interval limits
  (See procedure histo)
- **LB**: Index of first \( Y \)-column to be printed
- **UB**: Index of last \( Y \)-column to be printed
- **Y0**: Starting point of \( Y \)-axis
- **YS**: Scale factor for column length
The number of asterisks in the lines of the $K$-th (column) is

$$Y(k) - Y_0$$

$$YS$$

rounded to the nearest integer.

Example:

$Y_0 = 500$, $YS = 5$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>***</td>
<td>512.5 $\leq Y[k] &lt; 517.5$</td>
</tr>
<tr>
<td>***</td>
<td>$Y &lt; 502.5$</td>
</tr>
</tbody>
</table>

Parameters may be omitted by

1. Shortening the parameter list by cutting out from the right, or
2. By using as actual parameter a variable of another type than the one specified above.

Examples:

**Boolean Q**

$hprint (Y,Q,Q,UB) : X, LB, Y_0$ and $YS$ omitted

$hprint (Y) : X, LB, UB, Y_0$ and $YS$ omitted

$hprint (Y,Q,Q,Q,Q) : X, LB, UB, Y_0$ and $YS$ omitted

The effects of omitting parameters are:

1. $X$ : The intervals are instead given by the subscripts of $Y$.
2. $LB$ : The printout of $Y[k]$ -columns will start with the first $Y(k) > 0$. 
3. **UB**: The printout of \(Y[k]\)-columns will stop after the last \(Y(k) > 0\)

4. **YO**: YO is put equal to 0

5. **YS**: YS is automatically adjusted to make \(\text{max} (Y[k])\) consist of (3) columns of 100 asterisks.

The first and last columns of the Y-array will contain observations out of range. If these are empty, the mean (interval or subscript) value is printed after the histogram together with its standard deviation and the sum of the Y-columns.

At the extreme right of the printout the value of \(Y[k]\) is printed as a floating number together with the percentage which \(Y[k]\) represents of the sum of all \(Y[k]\)-s.


4. **procedure** arrinfo (text, code, A, COLMEAN, COLSD, COLMAX, COLMIN, COLRANGE, COLCORR, b);

    **text** is a **string**, code and b **integer**, A is a one- or two-dimensional **real array**, COLMEAN, COLSD, COLMAX, COLMIN and COLRANGE are one dimensional arrays of type **real**. COLCORR is a two dimensional array of type **real**.

A call on arrinfo will compute and print the user's choice among the following items:

1) The mean of the elements in each column of A.

2) The standard deviation from the mean of the elements in each column of A.

3) The maximum value of the elements in each column of A.

4) The minimum value of the elements in each column of A.

5) The range of the elements in each column of A.

6) The correlation coefficients between the columns of A.
The parameters have the following meaning:

text : heading to precede any printout

code : integer code specifying which of the six different items should be printed. The code is an additive binary code defined as follows:

32 : print COLMEAN (item 1)
16 : print COLSD  (item 2)
 8 : print COLMAX (item 3)
 4 : print COLMIN (item 4)
 2 : print COLRANGE(item 5)
 1 : print COLCORR (item 6)

A : the input array for which information should be computed.

COLMEAN : one-dimensional output array for item 1.
COLSD  : one-dimensional output array for item 2.
COLMAX : one-dimensional output array for item 3.
COLMIN : one-dimensional output array for item 4.
COLRANGE : one-dimensional output array for item 5.
COLCORR : two-dimensional output array for item 6.

b : the number of columns to be considered by the subroutine.

Parameters may be omitted as follows:

1) Shortening the parameter list from the right.

2) Shortening the parameter list from the left. If the parameter code is not present, a zero value will be assumed.

NOTE: the input array A must be present.

3) Using the integer constant 0 in place of any one of the six output arrays.

If COLMEAN is present, then COLSD must also be present.
Computation and printout is selected as follows:

1) If the parameter code indicates that an item should be printed or if the corresponding output array has not been omitted, then the item will be computed.

2) Printout will be given only for those items indicated by the parameter code.

The parameter A may be a one-dimensional array. In this case, COLCORR is undefined.

The following conditions will cause a run-time error message:

1. Each column of A contains only one element

2. The first parameter is a string and the second is not an integer.

3. The output parameters have incorrect type, dimension or length (except case integer 0).

Examples:

arrinfo ('ARRAY INFORMATION'; 62, ABC, 0, 0, MA, MI, RA, CO);
will cause the values of item 1,2,3,4 and 5 to be printed while the values of items 3,4,5 and 6 are returned to the user program through the array parameters MA, MI, RA and CO.

arrinfo (ABC, 0, 0, MA, MI, RA, CO);

As preceding example, but no printouts. This call is equivalent to

arrinfo ('', 0, ABC, 0, 0, MA, MI, RA, CO);
5. **Off-line plotting facilities**

Software have been written for the Norwegian Computing Center by Björn Kirkerud to produce paper tapes from SIMULA acceptable for off-line drawing by the KINGMATIC 1215 automatic drawing machine (produced by Kongsberg Våpenfabrikk, Kongsberg, Norway.)

A short summary of some of the subroutines available are given below:

1. **Basic Procedures**

   - **DEFPAGE** is used to define an origin, the scale and the size of the drawing.
   - **XPEN** and **YPEN** are functions defining the present X or Y coordinate of the pen.
   - **MOVE TO** move the pen without drawing.
   - **DLINE** draw a straight line.
   - **DCIRK** draw a circle.
   - **DOTON, DOTOFF** are used to control dotting of figures.
   - **STARTPLOT, STOPPLOT** is used to initialize and stop the drawing.
2. **High level procedures**

The "write" subroutine will accept "PLOTTER" as device. The text of the edited lines will be drawn instead of printed. The size of the text and the angle between the X-axis of the drawing machine and the text may be determined by the user. In addition to the symbols usually found on a high speed printer, special character sets containing small letters and foreign (Greek, Russian) alphabets will be provided.

The procedures SAVEPLOT, UNSAVEPLOT and DRAWSAVED form a triple that may be used to construct a library of pre-defined drawings that may be inserted in an actual drawing. The pre-defined drawing may be scaled to the required size, turned an angle and reflected.

The procedures XAKSE and YAKSE will draw axes for a coordinate system. More than one coordinate system may be used in a drawing.

LKURV1 and GKURV1 are used to draw curves. LKURV1 will use straight lines to connect successive points on the curve, while GKURV1 will use a parabolas.

DSTPL1 will draw a steeple diagram while DHIST1 will draw a histogram.

Examples on some drawings made using these routines are found at the end of this chapter.

The routines are now in restricted use at the NCC and full publications will be given in a separate report to be published in the near future.
0 - OBSERVED VALUE
X - PREDICTED VALUE (SMOOTHED)
SEPTEMBER 1967
STORE VOLUME (1000 TONS)
CHAPTER 9.

THE SIMULA SYNTAX.

9.1 The SIMULA Reference Language.

In the present chapter the syntax of a SIMULA reference language is defined. The language is somewhat more general than SIMULA as described in the preceding chapters. The concepts concerned are SIMULA blocks, activities, and sets.

The syntax is given as an extension of the ALGOL 60 syntax. Syntactic classes not defined here are understood to be those defined in the ALGOL 60 report. The following syntactic classes are understood to be redefined. The notation <as in the report> refers to the right hand side of the corresponding definition in the ALGOL 60 report. The section numbers are those of the report.

Section 3. Expressions.

<expression>::=<as in the report>|<element expression>|<set designator>

Section 3.4.1. Boolean expressions.

<relation>::=<as in the report>|<element expression>=<element expression>|<element expression>≠<element expression>

Section 4.1.1. Statements.

<unlabelled basic statement>::=<as in the report>|<SIMULA statement>

Section 4.6.1. For statements.

<for list element>::=<as in the report>|<element expression>|<element expression> while <Boolean expression>
Section 5. Declarations.
<declaration>::=<as in the report>|<activity declaration>

Section 5.1.1. Type declarations.
<type>::=<as in the report>|element|set

The following additional basic symbols are introduced.

<SIMULA basic symbol>::= SIMULA|activity|element|set|
activate|reactivate|at|delay|prior|
before|after|inspect|extract|
when|otherwise|new|none

9.2 Expressions.

9.2.1 Element expressions.
<activity identifier>::=<identifier>
<process designator>::=<activity identifier><actual parameter part>|new<activity identifier><actual parameter part>
<simple element expression>::= none|<variable>|<function designator>
<process designator>|<activity identifier>
<element expression>::=<simple element expression>
   <if clause><simple element expression>
   else<element expression>

9.2.2 Set designators.
<simple set designator>::=<variable>
<set designator>::=<simple set designator>
   <if clause><simple set designator>
   else<set designator>

9.3 SIMULA statements.

<SIMULA statement>::=<SIMULA block>|<scheduling statement>|<connection statement>
9.3.1 **SIMULA blocks.**

\[
\text{<SIMULA block> ::= SIMULA<unlabelled block>}
\]

9.3.2 **Scheduling statements.**

\[
\text{<activator> ::= activate|reactivate} \\
\text{<activation clause> ::=<activator><element expression>} \\
\text{<simple timing clause> ::= at<arithmetic expression>|} \\
\text{ delay<arithmetic expression>} \\
\text{<timing clause> ::=<simple timing clause>|} \\
\text{ prior} \\
\text{<scheduling clause> ::=<empty>|<timing clause>} \\
\text{ before<element expression>|} \\
\text{ after<element expression>} \\
\text{<scheduling statement> ::=<activation clause><scheduling clause>}
\]

9.3.3 **Connection statements.**

\[
\text{<connector> ::= inspect|extract} \\
\text{<inspection clause> ::=<connector><element expression>} \\
\text{<connection block> ::=<statement>} \\
\text{<connection clause> ::= when<activity identifier><do><connection block>} \\
\text{<simple connection part> ::=<connection clause>} \\
\text{<connection part> ::=<simple connection part>|} \\
\text{ otherwise<statement>} \\
\text{<connection statement> ::=<inspection clause><connection part>|} \\
\text{<connector><process designator><do><connection block>} \\
\text{<scheduling statement><connection part>} \\
\text{<activator><process designator><scheduling clause> do<connection block>}
\]

A symbol "when" or "otherwise" refers back to the nearest <inspection clause> or <scheduling statement>. This is sufficient to avoid ambiguity if a <connection block> is itself a <connection statement>. 
9.4 Activity Declarations.

\[ \langle \text{activity heading} \rangle ::= \langle \text{activity identifier} \rangle \langle \text{formal parameter part} \rangle ; \langle \text{value part} \rangle \langle \text{specification part} \rangle \]

\[ \langle \text{activity body} \rangle ::= \langle \text{statement} \rangle \]

\[ \langle \text{activity declaration} \rangle ::= \text{activity} \langle \text{activity heading} \rangle \langle \text{activity body} \rangle \]

9.5 Syntax Restrictions.

SIMULA, as defined in CHAPTERS 2-6 has the following syntax restrictions compared to the reference language.

1. The SIMULA basic symbols are only available within a SIMULA block, with the exception of the symbol "SIMULA".

2. SIMULA blocks must not be nested.

3. Activity declarations are only permitted in a SIMULA block head.

4. The parameter mechanism of an activity declaration is restricted to \langle type \rangle, \texttt{array}, and \langle type \rangle \texttt{array} parameters.

5. The \langle value part \rangle of an \langle activity heading \rangle is omitted.
   (Call by \texttt{value is assumed} for \langle type \rangle parameters, call by name otherwise.)

6. Assignment statements of the form
   \[ \langle \text{variable} \rangle ::= \langle \text{set designator} \rangle \]
   are not permitted.

7. The \texttt{set procedure} concept is excluded.

Obvious and possibly fruitful extensions can be made to SIMULA by removing some of the above restrictions. The removal of others would require a careful analysis of the semantics involved. The semantics of the unrestricted SIMULA reference language can be defined in more than one way.
CHAPTER 10.

THE UNIVAC 1107 SIMULA.

10.1 The Language.

The UNIVAC 1107 SIMULA is, excepting a few restrictions, implemented as an extension of the UNIVAC 1107 ALGOL. In particular the input/output and backing store facilities of the latter are available. For information concerning these topics the reader is referred to the "Programmers Guide" for the UNIVAC 1107 ALGOL. This document also provides the transliteration rules from the ALGOL 60 reference language and other necessary information.

The SIMULA basic symbols (see section 9.1) are implemented as reserved identifiers not usable for other purposes within a SIMULA block. The SIMULA library procedures for random drawing and data analysis (Ch.'s 7, 8) are considered declared in a block outside the program, whereas the other SIMULA procedures are treated as if declared in a dummy block immediately outside the SIMULA block. All procedure identifiers are therefore, by suitable redeclaration, usable for any purpose that the programmer may choose.

As in the UNIVAC 1107 ALGOL forward references to variables and other items (except local labels) must be resolved by corresponding "LOCAL" declarations in the relevant block heads.

10.2 Restrictions.

At the time of writing the following concepts are not implemented:

a. The own concept,
b. the call by value of a string parameter, and
c. STRING ARRAY in the sense of UNIVAC 1107 ALGOL.
Additional restrictions are:

1. A connection block may only refer to an activity already declared, or to the present one. (Forward references to activities by process designators are, however, resolved by a "LOCAL ACTIVITY" declaration in the SIMULA block head.)

2. Termination, passivation, or suspension of the currently active process is not in general tolerated during the evaluation of an expression, i.e. inside a function procedure. This is the case if the expression is part of
   a. an actual parameter,
   b. a subscript bound of an array declaration, or
   c. an element of a switch.

Suspension as the result of the direct scheduling of another event is permitted inside a function procedure, if the calling process is not terminated, cancelled, or reactivated before control returns.

10.3 Storage Requirement.

The data storage economy of UNIVAC 1107 SIMULA programs is far from the optimum. This is partly due to the way in which certain basic access mechanisms are implemented in the UNIVAC 1107 ALGOL system.

In order to achieve maximum storage economy the following rules should be observed within activities corresponding to a large number of processes.

1. Inactive periods within sub-blocks and procedures should be avoided.

2. The amount of data local to the outermost block, and the number of non-local variables referenced in that block, should be minimized.
3. The number of for statements and calls for activities and user defined procedures should be kept a minimum.

See also the next section.

10.4 Data De-Allocation.

The mechanism for de-allocating data at run time is fully automatic. However, an understanding of the principles involved will enable the programmer to have better control over the storage requirements of his program, if economy is required.

The main rules are the following:

1. The data structure local to a block, except the outermost block of a process, will remain only as long as the "local" sequence control (i.e. the main control or the reactivation point of the process in question) is within the block. The data may include simple variables, value parameters, and names on arrays and sets.

2. A process (including data local to a sub-block) remains only as long as there is at least one element referencing it.

3. An element remains only as long as there is at least one reference to it, from
   a. an element variable or value parameter, or a component of an element array,
   b. an element (through set membership),
   c. an event notice in the SQS, or
   d. a connection block.
4. An array remains only as long as it has a name. The name can be the declared array identifier or an exogenous array attribute of a process.

5. The elements of a set retain their set membership only as long as the set has a name. The name can be a simple declared set designator, a value set parameter, or a component of a set array. (The name need not be effectively referenceable, see below.)

6. An event notice can be removed from the SQS as the result of a sequencing statement, or when the (local) sequence control leaves an active process.

The rules 2-5 are put to effect by maintaining reference counts on processes and elements. A set head has a separate reference count for set designators referencing the set. There are, however, two cases where the reference count technique is not sufficient.

1. The value of an element expression, as residing in an accumulator immediately after the evaluation, is not reflected in the reference count on the element. An exception from rule 3 has to be made to prevent the de-allocation of the element value at exit from an element procedure, if the reference count indicates no reference to this element. (This is the case e.g. with all generative expressions.) Normally the element gets a reference assigned to it as the result of the statement containing the expression, but if this does not happen, and there is no other reference to it, the system does not get another chance to perform the de-allocation.
2. When certain "cyclic" data configurations are present, it may happen that an element with a non-zero reference count is not and can never become effectively referenceable through a computable element expression. Such an element can safely be de-allocated, even though it is not "out of the system" in the reference count sense.

As a simple example consider a process referencing itself through a local element variable X. Assume that there is no other reference to the element and no other element referencing the process. Evidently the only way of effectively referring to the element is to evaluate the expression X during an active phase of the process or in a connection block connecting it. But the process is passive or terminated, and not connected, because otherwise there would have been another reference to it, via the same or another element, from an event notice in the SQS or from a connection block, contrary to assumptions.

For these reasons a second de-allocation mechanism, the "storage clean-up" is brought to use whenever the available store is exhausted. The storage clean-up will locate all effectively referenceable parts of the system and make the remaining store available for use, if any.

Warning.

Since the storage clean-up may be brought in at unpredictable points of time, it should have no visible effect in the program. In particular every effectively referenceable element will retain its set membership. It follows that a set which can not be broken up by the reference count technique, e.g. a set local to a process referencing itself, will remain in the system as long as it contains at least one effectively referenceable element, even if it is not itself effectively referenceable as a set.
10.5 Operating Instructions

The UNIVAC 1107 SIMULA implementation consists of two distinct parts.

1. The **compiler** translates a SIMULA source language program to a program in object code. The compiler is a permanent extension of the UNIVAC 1107 ALGOL compiler.

2. The **run time system** is the collection of subroutines that may be referenced by a SIMULA object program. The run time system includes all SIMULA library procedures.

In the following a certain elementary acquaintance with the EXEC 11 monitor system is assumed. The reader is referred to the "UNIVAC 1107 EXEC II Manual" for details. The description is valid for an operating system where the SIMULA compiler resides on drum and the SIMULA library subroutines are on a tape called "SIMLIB". The appropriate operating procedure for the particular computing center should always be obtained before compilation and execution of a SIMULA program is attempted.

A SIM control card brings in the ALGOL compiler with the SIMULA extensions. The word "SIMULA" has been added to the compiler's list of reserved identifiers, so that SIMULA blocks can be recognized. Also the names of the random drawing and data analysis procedures of SIMULA have been added to the list of standard procedure identifiers of ALGOL. Within a SIMULA block all SIMULA concepts become available.

All options valid for an ALG control card are available and retain their usual meaning.

The SIMULA run time system resides on a magnetic tape. It must be part of the user's program complex file (PCF) on drum at the time of allocation of the absolute object program.
Some of the SIMULA routines will replace corresponding routines in the ALGOL run time system, which is a permanent part of the monitor library on drum.

Any ALGOL object program, subject to the restrictions mentioned in section 10.2, may be run with the SIMULA run time system. The execution will be slightly slower, but the available data store is increased, since the SIMULA storage allocation routines can utilize both memory banks.

The card deck of a simulation run could have the following typical layout:

```
V RUN ----
V ASF F = SIMLIB
V N XQT CUR
   IN F
   TRI F
V SIM PROG
   (Source language program)
V XQT PROG
   (Data cards)
V FIN
```

"PROG" can be any name chosen for the program. "SIMLIB" is the operational label adopted for the magnetic tape containing the SIMULA run time system. The CUR operations serve to include the latter in the user's PCF.

10.6 Initial values

Upon entry into a block variables (simple or subscripted) declared local to the block have the following initial values: zero, false, none, and blanks for arithmetic, Boolean, element, and string variables, respectively. Sets are empty.
CHAPTER 11.

ALGOL FUNDAMENTALS

In order to assist readers unacquainted with ALGOL in getting an impression of SIMULA, the basic ALGOL concepts are explained in this chapter. Readers are warned that this presentation is very incomplete. An ALGOL textbook must be consulted to get the knowledge necessary to write SIMULA programs.

11.1 Simple Variables and Declarations.

Variables to be used in ALGOL (and SIMULA) must be "declared": we write the name ("identifier") of the variable preceded by an underlined word stating the "type" to which the variable belongs.

ALGOL contains 3 types of "simple" variables: "Boolean", "integer" and "real". Boolean variables (named after the British mathematician Boole) may have one of the two "truth values", true and false. Integer variables may have a positive or negative integer or zero as values. Real variables may have any real number as values.

When variables are declared we write the type-name underlined, followed by the identifiers of the variables belonging to that type. The identifiers are separated by commas, and the list is ended by a semicolon.

Example:

Boolean tax paid, integer flightnr, qlength,
real baggage weight, arrivaltime, price,

ALGOL also contains subscripted variables, to be discussed in Section 11.6.
11.2 Statements and Programs.

An ALGOL program consists of
1. Declarations (e.g. of simple variables)
2. Statements describing actions involving the declared entities.

A program has an underlined `begin` as the first word (symbol), followed by the declarations and then the statements. The declarations and the statements are separated by semicolons and the program is concluded by the underlined word `end`.

Example:

The clerk at the check-in counter of an airport is told to report the total baggage weight as well as the weight of the checked baggage for a given flight. He reads the weight of the suitcases and then of the briefcase (the hand baggage) for each passenger as they turn up (the symbol `:=` means "becomes", and the statements are executed in sequence):

```
begin
real tot wt, checked, suitcases, briefcase;
tot wt := 0; checked := 0;
read(suitcases, briefcase);
    checked := checked + suitcases;
    tot wt := tot wt + suitcases + briefcase;
read(suitcases, briefcase);
    checked := checked + suitcases;
    tot wt := tot wt + suitcases + briefcase;
read(suitcases, briefcase);
write(checked, tot wt)
end;
```

(The same sequence is repeated for all passengers on the flight)
In the program we are allowed to operate upon the 4 real variables declared in Line 2. In Line 3 "tot wt" and "checked" weight are given 0 as initial value. Line 4 states that the values of "suitcases" (the weight of the suitcases) and "briefcase" now shall be read. The reading may involve a complex series of actions, which we only specify by the procedure-call read(------).

We only will assume that if "x" is a declared variable, then the execution of the procedure-call read (x) will assign a value to x from some source of information. In programs executed on computers, read (x) will usually mean that the value of x is read from a punched card. We will discuss procedures in section 11.10.

Since the suitcases are to be checked in, the weight of the checked baggage (initially zero) is increased by the weight of the suitcases (Line 5). We get the total weight by adding "suitcases" and "briefcase" to the present value of "tot wt" (Line 6).

Lines 7 - 9 repeat the actions of Lines 4 - 6, this time operating upon "suitcases" and "briefcase" for the next passenger. This sequence is now repeated till all passengers have arrived. Then a new procedure "write (------)" is called, recording the final values of "checked" and "tot wt".

11.3 Compound Statements.

Sometimes it is necessary to transform a sequence of statements to a single "compound" statement. This is done by bracketing the statement sequence between begin and end.
Example:

The statements in Lines 4 - 6 in the previous example may be written as a compound statement

```
begin read(suitcases, briefcase);
    checked := checked + suitcases;
    tot wt := tot wt + suitcases + briefcase end;
```

We may name this statement "check in", and rewrite the program in abbreviated form (for our own purpose, not in correct ALGOL):

```
begin declarations; tot wt := checked := 0;
    check in; check in; -----------; check in;
    write(checked, tot wt) end;
```

When we return to "procedures" in section 11.10 we will demonstrate how this use of the word "check in" may be made "legal" ALGOL.

11.4 Labels and go to - statements.

In the counter example Lines 4 - 6 ("check in") are repeated for every passenger. A language not taking advantage of this will lead to very long programs.

It is possible to mark an ALGOL statement by a "label": an (undeclared) identifier followed by a colon. This label then may be used in a go to - statement.

Example:

```
begin declarations; tot wt := checked := 0;
    next: check in; go to next end;
```

When "check in" of a passenger is completed, the same sequence is repeated by stating that the "check in" statement, marked by the label "next", shall follow the go to - statement.
This new version of the program is not able to transmit results of the computation, it goes in an infinite loop. The loop shall, however, only be executed as many times as there are passengers.

11.5 if - then - statements.

The ALGOL Boolean variables have two possible values: true or false.

The statement

"7 < 1"

is false. We name statements to which we may assign a truth value "Boolean expressions". Hence the value of the Boolean expression "7 < 1" is false.

ALGOL also contains statements executing the standard operations of two-valued logics:

\[ \land : \text{logical "and"} \]
\[ \lor : \text{logical "or"} \]
\[ \neg : \text{negation} \]
\[ \Rightarrow : \text{implication} \]
\[ \equiv : \text{equivalence} \]

Example:

Let us assume that the clerk computes the baggage weight. If the limit is 20 kilos, excess baggage freight must be paid. Then, for a given passenger, the postulate "money left" may be true or false.
We introduce the variables \texttt{real} excess wt and \texttt{Boolean} money left. The course of further action will then be dependant on the Boolean expression.

\[(\text{excess wt} > 0) \land (\neg \text{money left})\]

In an "if - statement", two alternate courses of action may be prescribed, depending on the value of a Boolean expression:

Let B be a Boolean expression and S1 and S2 two statements. We may write

\[\text{if B then S1 else S2;}\]

implying that S1 shall be executed if B has the value \texttt{true}, S2 if B has the value \texttt{false}. If the alternative to S1 is "no action", we may write

\[\text{if B then S1;}\]

Examples.

We now may complete the counter example. Let \texttt{integer} nr passengers, checked passg denote the total number of passengers on the flight and the number of passengers already checked in.

\begin{verbatim}
begin Declarations; tot wt := checked := 0; checked passg := 0;
read(nr passengers);
next: check in; checked passg := checked passg + 1;
    if checked passg < nr passengers then go to next;
write(checked, tot wt)
end;
\end{verbatim}

Returning to the excess weight situation, we may want to state

\[\text{if (excess wt} > 0) \land (\neg \text{money left}) \text{ then go to home else go to aircraft;}\]

where "home" and "aircraft" are two labels in other sections of the program.
If we want to execute a sequence of statements in each "branch" of the \texttt{if} - statement, these sequences must be written as compound statements. \textit{S}1 and \textit{S}2 must each be one statement, but may be a simple or a compound statement.

11.6 Arrays.

\texttt{Boolean, integer} and \texttt{real} variables may be subscripted and form \texttt{arrays}. In their declarations the ranges of variation of the subscripts are specified:

\begin{verbatim}
Boolean array open counter \[i:6\] ;
integer array q length \[i:6\] ;
\end{verbatim}

We now refer to the queue length of counter \textit{nr} 5 by \texttt{q length [5]}, and may write e.g. (\textit{k} is an \texttt{integer} variable):

\begin{verbatim}
if open counter \[k\] then q length \[k\] := q length \[k\] + 1;
\end{verbatim}

assuming that \(1 \leq k \leq 6\).

Arrays may also be \textit{multidimensional}. Let us assume that all airports are numbered 1, 2, \ldots, \textit{N}. Then the destination of a passenger may be specified by an \texttt{integer} between 1 and \textit{N}.

If we need a table of ticket prices, \textit{1st} and Tourist Class (regarded as \textit{2nd} Class) this is done by introducing a \texttt{real array} (or just \texttt{array}):

\begin{verbatim}
array ticketprice [1 : 2, 1 : N] ;
\end{verbatim}

The ticket price on Tourist Class to Airport \textit{nr.} 122 may then be referred to by

\begin{verbatim}
ticketprice [2, 122].
\end{verbatim}
11.7 for - statements.

The ALGOL for - statements are flexible tools for specifying repetition of statements.

Example 1:

The repetition in the counter example may be written:

\[
\text{for checked passg := 1 step 1 until nr passengers do check in;}
\]

where "check in" is the above compound statement.

The for - statement tells us that "checked passg" shall be given the initial value 1 and "check in" be executed. Then "checked passg" shall be increased by 1 ("stepped up") and "check in" once more executed. When checked passg = nr passengers "check in" is executed for the last time and then the next statement is executed.

Example 2:

Another type of for - statement is the following:

\[
\text{begin integer tot q, k; integer array q length [1:6] ;}
\]

\[
\text{tot q := 0;}
\]

\[
\text{for k := 1, 3, 6 do tot q := tot q + q length [k] ;}
\]

\[
\text{end;}
\]

k is successively given the values 1, 3 and 6, and for each of these values the statement after do is executed.
Example 3:

A third alternative in the counter example is to write

```
begin ---------------------
    checked passg := 0;
    for checked passg := checked passg + 1 while
        checked passg ≤ nr passengers do check in;
----------------------------- end;
```

11.8 Blocks.

An ALGOL "block" is a series of declarations followed by one or more statements. Each statement may be simple or compound. A block starts with `begin` and is concluded by `end`.

The declaration part of a block is called the "block head".

Example:

The passenger at the counter has to pay an excess freight for his baggage if the weight exceeds 20 kilos. For each kilo of excess weight he has to pay 1% of the 1st Class ticket price to his destination. We will assume that there are 500 possible destinations.

```
begin real suitcases, briefcase, excess freight;
    integer destination; array ticket price [1 : 500];
-----------------------------
------------------------------
    if suitcases + briefcase > 20 then
        begin real excess wt;
            excess wt := suitcases + briefcase - 20;
            excess freight := excess wt × ticketprice [destination]/100
        end
    else excess freight := 0;
------------------------------
end;
```
Here the whole program is a block, and the compound statement after "then" also is a block with its own block head and statements, inside the program block.

A block is itself a statement and therefore may contain other blocks, "inner blocks".

Variables declared within a block are said to be "local" to this block.

Variables local to a block are available for use in all statements inside the block, including inner blocks.

Variables local to a block are not available for use outside the block.

In the example "suitcases" and "excess freight" are declared in the outer block and hence may be used in the inner block. "excess wt" only exists within the inner block.

These properties of ALGOL blocks are very useful, since large data structures only needed in a small part of the program do not occupy space in the computer memory when other parts are executed, if the part where these data are needed is written as a block. Also the same name may be used in different blocks B1 and B2 for different purposes without confusion. If B1 is an outer block to B2, the "outer variable" is not accessible within B2. The common name has its "local" significance within B2.
The block structure of ALGOL is fundamental for its usefulness as component of a simulation language.

11.9 ALGOL programs.

An ALGOL program is a block or a compound statement having no outer block and not being a part of another compound statement.

11.10 Procedures.

In the counter example we have used the abbreviation "check in" for the compound statement

\[
\begin{align*}
\text{begin} & \quad \text{read(suitcases, briefcase);} \\
& \quad \text{checked} := \text{checked} + \text{suitcases} ; \\
& \quad \text{tot wt} := \text{tot wt} + \text{suitcases} + \text{briefcase end};
\end{align*}
\]

The use of this abbreviation may be made "legal" by declaring "check in" as a procedure.

Together with the declarations of variables in the block head we write a "procedure declaration":

\[
\begin{align*}
\text{procedure} & \quad \text{check in;} \\
\text{begin} & \quad \text{read(suitcases, briefcase);} \\
& \quad \text{checked} := \text{checked} + \text{suitcases} ; \\
& \quad \text{tot wt} := \text{tot wt} + \text{suitcases} + \text{briefcase end};
\end{align*}
\]

This being done, it is permissible in ALGOL to use the statement "check in" in the above manner. The statement is usually said to be a "procedure call" (correctly: "a procedure statement").

In the above procedure declaration, the first line

\[
\begin{align*}
\text{procedure} & \quad \text{check in;}
\end{align*}
\]

is called the "procedure heading". The compound statement following the heading is called the "procedure body". In general the procedure body is a block, possibly having its own declarations.
A procedure statement ("call") prescribes an execution of the procedure body at this place in the program.

The procedure "check in" results in operations on the same variables each time it is called. The read-procedure mentioned earlier may be used to assign values from an input device to any variable. The variable which shall get a value is given as a parameter to the call:

\[ \text{read (nr passg), read(suitcases).} \]

The procedures for input and output of data are not standardized in ALGOL and have to be specified for the machine which is used for the computation.

Certain commonly used mathematical functions are, however, considered to be part of the language, e.g. the sine and cosine functions, the square root, the exponential function \(e^x\).

The calls for evaluation of \(e^x\) and \(\sqrt{x}\) (where \(x\) is a real variable) are written

\[ \text{exp(x), sqrt(x)} \]

If we in a computation need repeated evaluation of a function not being part of ALGOL, we may declare a function procedure in our program.

**Example.**

We want to declare the hyperbolic cosine \(\cosh(x)\) as a procedure. \(\cosh(x)\) has a real value, and

\[ \cosh(x) = \frac{1}{2} (e^x + e^{-x}) \]

**real procedure** \(\cosh(x)\); \(\text{real } x;\)

\[ \cosh := (\text{exp(x)} + \text{exp(-x)}) / 2; \]
In the procedure heading we have specified that $x$ is a real parameter, which means that the "actual parameter" given in a call for the procedure should be some real expression. Such specifications are an optional part of the language, but many compilers require specification of all parameters, since this information makes it possible to generate more efficient object programs.

A call for a function procedure is called a "function designator". It is an expression whose type is specified by the first word of the procedure heading. The function designator "cosh(y)" evaluates the hyperbolic cosine of the real variable $y$, as stated in the procedure declaration above. $y$ is the actual parameter of the call. The actual parameter can in this case be any arithmetic expression; $\cosh(\log(y) + 0.5/z[1])$ gives the hyperbolic cosine of the value of the expression.

There is an important difference between parameters to the "read" and "cosh" procedures. The call read(y) assigns a value to the variable $y$, whereas the cosh procedure only uses the value of the given actual parameter for its own purpose.

Formal parameters of the second kind can be included in a "value specification" in the procedure heading:

```
real procedure cosh(x); value x; **al x; ----- 
```

A value specification improves the efficiency of the object program and should be given if possible.

If a formal parameter is included in a value specification, it is said to be "called by value", otherwise it is "aid to be called by name".

A parameter called by value is similar to a local variable, except that it has an initial value defined by the actual parameter of the call. An assignment to a value parameter has no effect outside the procedure body.
A parameter called by name is considered to be replaced by whatever expression is given as the actual parameter of the call. If an assignment is made to a name parameter, the actual parameter must be a variable. This variable receives a value as a result of the procedure call. Such a parameter is often called an "output parameter", since it transmits output from the procedure.

A procedure can refer directly to variables and other items non-local to the procedure body, i.e. to items local to the block containing the procedure declaration, or to outer blocks. If the value of a non-local variable is changed within the procedure body, this is called a "side-effect" of the procedure.

As an example on the general procedure concept we will formulate a procedure for evaluating the polar coordinates \((r, \theta)\) of a point whose cartesian coordinates are \((x, y)\). \((x\text{ to the } n\text{th power is written } x^n\text{ in ALGOL.})\)

\[
\text{procedure} \quad \text{polar} \quad (x, y, r, \theta); \quad \text{value} \quad x, y; \quad \text{real} \quad x, y, r, \theta; \\
\text{if} \quad x = 0 \land y = 0 \quad \text{then} \quad r := \theta := 0 \\
\text{else begin} \quad r := \sqrt{x^2 + y^2}; \\
\quad \theta := \arctg(x, y) \quad \text{end;}
\]

where \(\arctg\) evaluates the arc in a suitable range.

Clearly \(r\) and \(\theta\) must be called by name, since they are output parameters, whereas \(x\) and \(y\) can be called by value.

The procedure statement

\[
\text{polar} \quad (3, 7, f, g)
\]

will assign the polar coordinates of the point \((3, 7)\) to the real variables \(f\) and \(g\).

The call

\[
\text{polar} \quad (z + 1, w - u, f, g)
\]

will assign to \(f\) and \(g\) the polar coordinates of the point whose cartesian coordinates are the current values of the expressions \(z + 1\) and \(w - u\).
CHAPTER 12.

A SIMPLE SIMULA DESCRIPTION

12.1 A Simple Disease System.

As a first example on a SIMULA system description, let us consider a simple disease system:

Infections occur in a limited population with a time distribution given by a Poisson distribution with parameter "real p". Infected persons do not infect other persons, and they become immune if they recover from the disease.

When a person is infected, no symptoms appear the first 3 days. In the following 7 days (days 4 through 10) the probability of death is given by an array:

\[
\text{array mortality} \ [4 : 10]
\]

As mentioned in section 1.4 every entity which carries out actions and/or is a carrier of data in SIMULA will be called a process. All processes characterized by the same data structure and having the same pattern of behaviour (operator rule) are said to belong to the same activity.

Obviously we have to introduce the infected persons as processes. Since the course of development of the disease is the same for all persons, they all belong to one activity:

\[
\text{activity infected person;}
\]
Apart from the "main program" taking care of the initialization of the system, the introduction of new "infected persons" and final analysis of the results of the disease, we have no other entities executing actions or carrying data, and consequently no more activities.

We start our formal description of the system by writing

```simula
SIMULA begin
```

(Line 1)

telling that SIMULA concepts are to be used.

"In the next line we declare the "system variables", variables which shall be available for use everywhere in the system: the size of the population (named "population"), the number of not yet infected persons left in the population (named "nr uninfected"), the parameter "p" of the Poisson disease occurrence distribution, the table of real numbers giving the mortality on the n-th day of the disease (named mortality[n]), and finally two integer variables U1 and U2 used in selecting the streams of random numbers to be used in the random drawings performed (see section 7.1).

U1 and U2 are introduced because of the possible simulation of the system. They have no interpretation in terms of the stem which is described.

The declarations of system variables then become:

```simula
integer population, nr uninfected, U1, U2;
real p; array mortality [4 : 10];
```

(Line 2, Line 3)

After the disease has disappeared we want to find out how many persons died and how many were cured. For this we need to establish two lists of references to these categories of persons. As mentioned in section 1.4 the SIMULA `sets` will
serve this purpose. Since all processes must be able to refer to these sets, they are declared as "system sets":

\texttt{set \text{dead}, \text{cured};} \quad \text{(Line 4)}

Notice that the set declarations introduce \texttt{empty} sets, whereas the values of variables are \texttt{undefined} till an assignment of a value is made.

We may now start the description of the "infected person" activity by writing:

\texttt{activity \text{infected person};} \quad \text{(Line 5)}

We have to keep track of how long the person has been infected, since this determines the mortality of the disease at any given stage. Hence the infected persons are characterized by an \texttt{integer} attribute "day":

\texttt{begin \text{integer} \text{day};} \quad \text{(Line 6)}

(The \texttt{begin} indicates that the description of the endogenous attributes and operation rule of the activity starts here).

When stating the operation rule for a SIMULA activity we are able to take a completely "local" view, and concentrate upon the sequence of actions which may be carried out by an individual process belonging to the activity. Since all actions are carried out by processes during active phases (events) and a process either belongs to an activity or is the unique "main program" process, all actions in the system will be described by giving the operation rules of the activities and the main program. The proper interlacing of events will be achieved by the sequencing statements (see \texttt{CHAPTER 4}).

During the first active phase of an infected person the following statement is executed:

The number of uninfected persons is reduced by 1, since a new infected person has become active.

\texttt{nr \text{uninfected} := nr \text{uninfected} - 1;} \quad \text{(Line 7)}
No more actions are to be executed until 3 days have passed by. We shall "hold" the sequence of actions for 3 days, the process "suspending" itself for 3 days. This is described by the statement

\[ \text{hold}(3); \]  

(Line 8)

When the 3 days have expired, actions and decisions have to be made the following 7 days:

\[ \text{for day := 4 step 1 until 10 do} \]  

(Line 9)

Each day there is a probability, mortality [day], that the person dies. If this happens, a reference to him is included in the set "dead" and all further actions are stopped, he is "terminated".

In other words there is a probability, mortality [day], that these actions occur. This is equivalent to making a random drawing with probability mortality [day] of getting true as result, and only execute the actions if this turns out to be the outcome. For this purpose we may use the "draw" - procedure described in section 7.2:

\[ \text{if draw(mortality [day], U1) then} \]
\[ \begin{align*}
\text{begin include(current,dead);} \\
\text{terminate(current)} \\
\text{end}
\end{align*} \]  

(Line 10)

(Line 11)

"current" always references the process which currently is active, that is, the process in which the reference is made. From the process' point of view, "current" means "myself".

If the result of the drawing is false, further actions are suspended till the next day:

\[ \text{else hold(1);} \]  

(Line 12)
Then, if the previous day was not the 10th, "day" is increased by 1 and Lines 10-12 are repeated.

If the previous day was the 10th, we have completed the for - statement starting at Line 9, and we proceed to day nr. 11. Now the infected person has survived and is cured and immune, and he may include a reference to himself in the "cured" set:

\[
\text{include(current, cured)}\quad (\text{Line 13})
\]

This is the last action relevant to the disease performed by the infected person, and we may conclude the operation rule by

\[
\text{end;}\quad (\text{Line 14})
\]

The infected person has no more actions prescribed for himself. He is terminated, but still remains in the system as a data structure, since he is a member of a set.

Till now we have only made declarations: of system variables and sets, of data structures and operation rules of processes which may appear in the system. No actions have been executed.

The sequence of statements of the SIMULA block, following the declarations, is the operation rule of a unique process, the "main program", always present in a SIMULA system description. Since the first statement after the declarations always is the first statement executed within a block, the first active phase of the main program always is the first event within a discrete event system as described by SIMULA. It is bound to occur, and as first event it is assigned the system time zero.

If other processes are to appear, at least one has to be generated by the main program and the main program must have at least one inactive period if events pertaining to other processes shall be executed.
As the process having the "first event", the main program is used for setting up the initial state of the system. It may also, if desired, be used as a "monitoring" process and for final analysis of information collected during the operation (or simulation) of the system.

The first we have to do to set up the system is to read the parameter values from an input device:

\[
\text{read(population, U1, U2, p, mortality); } \quad \text{(Line 15)}
\]

Then the initial value of "nr uninfected" is set

\[
r \text{ nr uninfected := population;} \quad \text{(Line 16)}
\]

We now initiate the first "infected person"-process, and since we are to repeat this statement we mark it with a label:

\[
\text{infect; activate new infected person;} \quad \text{(Line 17)}
\]

This statement generates a new "infected person"-process by the "generative" expression

\[
\text{new infected person}
\]

and activates this process immediately. By "immediately" we mean that the main program suspends its own actions to allow the just generated "infected person"-process to execute its first active phase, described in Lines 5-8. This is called "direct scheduling" (see CHAPTER 4).

When this process suspends itself (Line 8), the main program immediately resumes its actions in a new active phase.

This active phase consists of only one statement, suspending the actions of the main program for the time interval between this and the next appearance of an infected person. The length of this interval has a negative exponential distribution, and we may write:

\[
\text{hold(negexp(p, U2)); } \quad \text{(Line 18)}
\]
If there are any more uninfected persons, we repeat the infection:

    if nr uninfected >0 then go to infect;  \hspace{1cm} (Line 19)

If all persons are infected, we proceed to the next statement which has to be

    hold(11)  \hspace{1cm} (Line 20)

since no analysis should be made till the last infected person has died or become cured. Then we may write out the results:

    write(population,cardinal(dead),cardinal(cured))  \hspace{1cm} (Line 21)

The procedure "cardinal" gives the number of references ("elements") in a set. This is the last statement, the SIMULA block and the SIMULA description is concluded by

    end;  \hspace{1cm} (Line 22)

The complete description becomes:

\begin{verbatim}
SIMULA begin
integer population, nr uninfected, U1, U2;
real p; array mortality [4 : 10];
set dead, cured;
activity infected person;
begin integer day;
    nr uninfected := nr uninfected - 1;
    hold(3);
    for day := 4 step 1 until 10 do
        if draw(mortality [day], U1) then
            begin include(current,dead); terminate(current) end
        else hold(1);
            include(current,cured)
    end;
    read(population, U1, U2, p, mortality);
    nr uninfected := population;
    infect: activate new infected person;
end;
\end{verbatim}
12.2 Details of the Element and Sequencing Procedures.

In this section we will discuss in detail the sequencing of events in simulations generated by SIMULA system descriptions by using the example in the last section.

All events which are scheduled, on which we have information, are represented by an "event notice" in a set, "the sequencing set" (SQS).

An event is determined by the system time at which the event is to occur and by the process which is to be active. The event notice must contain these two informations: a system time reference (TR) and a process reference. However, since all references to processes in SIMULA are indirect, through elements the event notice instead of a process reference contains an element reference (ER). The element referred to will in turn refer to the process.

Thus the format of an event notice is

(\text{TR}, \text{ER}).

The event notices in the SQS will at any time be ordered according to increasing value of the TR. The event notice lying in the front of the SQS refers to the currently active process, the "current" process. When the current event is completed, its event notice is removed. The event notice which occupied the second position now becomes the first, and the next active phase of its (indirectly) associated process becomes the new "current event".

Since the main program is the only process existing initially and it is bound to have an active phase, the initial contents of the SQS is an event notice referring an element which in turn refers the main program. Let us name the element E(MP). The system time is in SIMULA set equal to zero at the start of
system operation. The contents of the SQS then is

\[(0, \text{E(MP)})\]

This event causes the first 3 statements of the main program to be executed (Lines 15-17).

In Line 17, the expression "new infected person" creates a new process, belonging to the activity "infected person". Let us name this process \( P_1 \) in our discussion.

The reactivation point (RP) is placed behind the declarations of attributes. This implies that \( P_1 \) when activated will start its first active phase by executing the first statement of its operation rule. (See CHAPTER 2). During the active phase the RP is not defined. When it is completed, the RP is placed behind the statement concluding the active phase, thus defining the action whereby a new active phase of this process shall start.

Together with \( P_1 \) an element referring to \( P_1 \) is generated. Let us name this element \( E(P_1) \).

As described in CHAPTER 3, an element contains 3 references. One reference specifies a process, the two others serve to specify a set membership, if any. Since the SIMULA sets are ordered and cyclical, these two references specify a successor and a predecessor in the set. The format of an element then is

\[(\text{successor, predecessor, process})\]

The element of the main program contains

\[(\text{none, none, main program}),\]

and the element of \( P_1 \) contains

\[(\text{none, none, } P_1)\]
In this discussion, the SQS will be ordered as follows:

SQS
E1
E2
E3
·
En

E1 is the current event, at the front of SQS. E2 is the next event to E1 etc.

The expression "new infected person" is preceded by the word activate. This implies that P1 shall have its first active phase at the current system time and with priority before all other events, even the current one. ("Direct scheduling": "activate E" is equivalent to "activate E delay 0 prior". See CHAPTER 4).

To achieve this, the current event is not cancelled, it only suspends itself by placing the event notice generated by "activate new infected person" in front of itself. "activate new infected person" then generates an event notice

\[(0, E(P1)),\]

and the contents of the SQS becomes

\[
\text{SQS:}
\]

\[(0, E(P1)),
(0, E(MP)).\]

The consequence is, according to the rules stated, that the first active phase of P1 becomes the current event, and P1 executes the first statements of its operation rule (Lines 7-8). ("new infected person" already has had the effect of generating the process and its attribute "integer day").
hold(3) in Line 8 concludes the first active phase of P1, and at the same time schedules a next active phase for itself to occur 3 time units later, it "suspends" itself. ("hold(3)" is equivalent to \texttt{reactivate current delay 3}")

This scheduling is done by generation of an event notice \((0 + 3, E(P1))\), which is inserted at its proper place in the SQS. Since the SQS is ordered according to increasing TR, the SQS becomes

\[
\text{SQS:} \\
(0, E(MP)), \\
(3, E(P1)).
\]

The event notice \((0, E(P1))\) has disappeared because its associated event is completed.

The main program once more becomes the current process, and the system time still is zero. The only action in this event is \"hold(negexp(p,U2))\" (Line 18). The main program suspends itself. Let us assume that the random drawing \"negexp(p,U2)\" gives 2.5 as its result. The hold-statement then passivates the main program and generates the event notice \((0 + 2.5, E(MP))\). We get

\[
\text{SQS:} \\
(2.5, E(MP)) \\
(3 , E(P1)).
\]

Once more the main program becomes active. The system time is stepped forward to the TR of the current event, 2.5, and the statement of Line 19 is executed. We assumed that the size of the population is large, and we will then execute Line 17 once more: the main program creates a new event notice referring to a new element \(E(P2)\) which refers to the new process \(P2\) (generated by the execution of \"new infected person\"), and suspends itself.
The SQS becomes

SQS:

(2.5, E(P2))
(2.5, E(MP))
(3, E(P1))

P2 executes Lines 7-8 of its operation rule (which it shares with P1 and all other "infected person"-processes to be generated) and generates the event notice (2.5 + 3, E(P2)) which gives the new contents of the SQS:

(2.5, E(MP))
(3, E(P1))
(5.5, E(P2))

The main program now executes the hold-statement. Let us assume that the result of the random drawing is 2, which gives

SQS:

(3, E(P1))
(4.5, E(MP))
(5.5, E(P2))

P1 now gets its second active phase. The reactivation point of P1 indicates that the for-statement of Line 9 now shall be started: "day" is put equal to 4 and the random drawing of Line 10 is executed. Let us assume that the outcome is "false". Then "hold(1)" is executed and the SQS becomes

(4, E(P1))
(4.5, E(MP))
(5.5, E(P2))

P1 gets its 3rd active phase: "day" is increased by 1 to 5 and once more a random drawing is performed. Let us assume that the outcome this time is "true". Then the statement "include(current, dead)" shall be executed. "current" now is P1, or rather E(P1), and the set membership of the element must be modified:

E(P1) shall be included as the last element of the set "dead". This set has till now been empty: it has only contained its "set head", which has been its own successor and
predecessor:

head(dead): (head(dead), head(dead), "no process")
(as always the process reference of a set head is to "no process")

E(P1) shall become the last element of "dead":
head(dead) shall become its successor element. Since there are
as yet no other elements, the elements of "dead" after the inclusion are

head(dead) : (E(P1),E(P1), "no process")
E(P1) : (head(dead),head(dead), P1)

The statement "terminate(current)" ends the active phase of
P1 without leaving a reactivation point or an event notice.
P1 is terminated, it only exist in the system as an element of
the set "dead" and a carrier of the variable "day", having the
value 5.

We have

SQS:
(4.5, E(MP))
(5.5, E(P2))

The main program generates P3, E(P3) and the directly sche-
duled event notice (4.5, E(P3)):

SQS:
(4.5, E(P3))
(4.5, E(MP))
(5.5, E(P2))

The first active phase of P3 gives

SQS
(4.5, E(MP))
(5.5, E(P2))
(7.5, E(P3))
Let us assume that the random drawing in Line 17 gives 1.5 as result:

\[ \text{SQS:} \]
\[ (5.5, \text{E(p2)}) \]
\[ (6, \text{E(MP)}) \]
\[ (7.5, \text{E(P3)}) \]

Now P2 enters its 2nd active phase at the for-stateme nt at system time equal to 5.5. Let us assume that the random drawing already at day=4 gives true as outcome. Then E(P2) shall be inserted as the last element of "dead". This implies that it gets head(dead) as its successor and E(P1) as its predecessor. The references of the other elements are modified accordingly, and the set becomes:

- head(dead): (E(P1), E(P2), "no process")
- E(P1): (E(P2), head(dead), P1)
- E(P2): (head(dead), E(P1), P2)

Then P2 is terminated and only exist in the system as an element of "dead" and a carrier of the information "day=4".

In this way, the sequencing is continued. Before we leave the example, let us consider some other situations.

Let us assume that

\[ \text{SQS:} \]
\[ (15, \text{E(MP)}) \]
\[ (16, \text{E(Pn)}) \]
\[ (17.8, \text{E(Pn+1)}) \]
\[ \vdots \]
\[ \vdots \]

and that the main program executes its hold-statement giving the event notice (16, E(MP)). According to the definition of the hold-statement, the generated event notice has no priority. Therefore it is inserted after all other event notices with TR = 16 already in the SQS, which becomes
SQS:
(16, E(Pn))
(16, E(MP))
(17.8, E(Pn+1))
;

If we in Line 18 instead had written
\texttt{reactivate current delay negexp(p,U2)prior}
the event notice would have had priority over other event
notices with the same TR:

\begin{verbatim}
SQS:
(16, E(MP))
(16, E(Pn))
(17.8, E(Pn+1))
\end{verbatim}

Notice that we have to write \texttt{"reactivate"} in this case: the
main program is already active and therefore \texttt{"activate"} will
have no effect.

\texttt{"reactivate E ---"} removes the existing event notice, if any,
and substitutes the new one generated by the statement.

Let us consider the situation when an "infected person"-process,
Pn, has survived the 10th day. He executes the final \texttt{hold(1)-}
statement, and in his next active phase the statement "include
(current, cured)" is executed. This inclusion follows the
same rules as described for the "dead" set above. No terminate-
statement is specified, but the process is automatically termi-
nated since no other statement follows the include-statement.
The local "sequence control" of the process' actions has "left
through the final \texttt{end}" of the operation rule.

When all members of the population are infected, the statement
\texttt{hold(11) (Line 20)} of the main program is executed. The re-
sulting event notice will have a larger TR than any other EN
already generated or to be generated in the system operation.
This implies that this event notice will become and remain the last one in the SQS. When it finally is the only one not executed, the main program executes its last active phase, Line 21, and the main program becomes terminated, control leaves through its final end.

Now the SQS is empty, and no more actions will occur in the system. If the SIMULA description is an inner block in an ALGOL program, control will proceed to the next statement in the outer block. All entities declared and generated within the SIMULA block will disappear. However, variables declared in the outer block may have been operated upon within the SIMULA block, which may in this way transmit information to the outer block.

12.3 Details of Scanning and Connection.

As stated above, the "infected person"-processes will end up as terminated members of the sets "dead" or "cured". For those who are elements of "dead", the integer attribute "day" will specify at which day after infection the person died. This we may want to use to establish a histogram of the days of death. Let "nr dead [4 : 10]" be an integer array, declared in the SIMULA block head.

We may perform the calculations inside the operation rule of the "infected person" activity, but we will now show how we may use the connection facilities of SIMULA for this purpose (see CHAPTER 5).

All the "dead" processes are members of the "dead" set. Each of these processes carry the relevant information in the attribute "day". If we want to compute the statistics during the last active phase of the main program, the use of the identifier "day" has no meaning: "day" is not declared in the SIMULA block head, and we have as many values of "day" as we have processes in the system.
In order to get all these values available, one at a time, we must be able to refer to each individual process, and make its attribute available to be operated upon.

If E is an element referring to a process P belonging to the activity A, and S is a statement, the statement

\[ \text{inspect E when A do S;} \]

will "connect" the process P so that its attributes are available in the statement S. There is no ambiguity now, since we have specified an individual process through the element E, and it is the individual values of the attributes of this process P we get access to.

To use this device, we also must be able to let E refer to all members of the set "dead" in succession, we must "scan" the set. This is done by using an "element variable", a variable taking on specific elements as values. Let us call this variable "pointer" and declare it in the SIMULA block head by

\[ \text{element pointer;} \]

We start the scanning by letting "pointer" refer to the element head(dead):

\[ \text{pointer: = head(dead);} \]

The scanning is achieved by repeated substitution of "pointer" by its own successor:

\[ \text{pointer: = suc(pointer)} \]

After the first substitution "pointer" will have the "first" element of "dead" as its value, after the second substitution the second element etc.
Since the SIMULA sets are cyclic, such a sequence of substitutions would never stop. Therefore we have after each substitution to test whether the element now referred to by "pointer" itself refers to a process. For this purpose we use the Boolean procedure "exist". exist(pointer) will have the value true if pointer refers to a process through its element value, false if its element value has no process reference.

Since only the head of a set has no process reference, we may continue the substitution as long as exist(pointer) has the value true.

After each substitution we connect the process referred to and use its "day" attribute for calculations.

The lay out of the SIMULA program becomes:

element pointer; and integer array nr dead [4 : 10];

are declared in the SIMULA block head.

After the statement hold(11) (Line 20) we give all variables in nr dead zero as initial value (k is an integer variable also declared in the SIMULA block head):

\[
\text{for } k: = 4 \text{ step 1 until 10 do nr dead } [k]: = 0;
\]

Then pointer is given its initial value

\[
\text{pointer : = head(dead);}
\]

and the scanning, connection and calculation is done by

\[
\text{for pointer : = suc(pointer) while exist (pointer) do inspect pointer when infected person do }
\]
\[
\text{nr dead } [\text{day}]: = \text{nr dead } [\text{day}] + 1;
\]

(The last statement simply increases by 1 the number of dead for the day of death of the person connected).
As last statement we may specify a printout:

    write (population, nr dead, cardinal(cured));

which will print all components of the array.

The "inspect"-statement will not affect the set membership of the elements referred to. Another connection statement

    extract E when A do S;

will connect the process referred to by E, but at the same time remove E from the set to which it belongs, if any.

The "extract" statement may also be used in our case, since it is irrelevant whether the set membership is kept or not. However, the substitution procedure must be a different one, since suc(pointer) is none after "pointer" has been extracted from the set "dead".

When "pointer" has been extracted and the necessary calculations made, we want to get the next process which now, after the previous extraction, is the first element of the set. In this case we may omit the statement "pointer = head(dead)"; and instead write

    for pointer: = first(dead) while exist(pointer) do
        extract pointer when infected person do
        nr dead [day]: = nr dead [day] + 1;

Notice that the element extracted looses its only reference in the system, through the set "dead", by the extraction. On the other hand it gets another reference added through the connection. After the connection statement is completed this reference vanishes. No other reference is created in the connection statement, it is not possible to refer to the element any more: it disappears from the system. Since this element is the only one referring its process, the process also disappears.
batch of units of a given order waiting in the queue, the next order is tried. The last units of an order are accepted as a batch, even if the number is less than the ordinary batch size. If a machine finds no acceptable batch in the product queue, it will wait until more units arrive.

Although the individual pieces of product are "units", a unit will not be treated as an individual item in the present model. For a given order and a given step in its schedule, i.e. machine group, we define an opart (order part) record to represent the group of units currently involved in that step. The units are either in processing or waiting to be processed the corresponding machine group.

An order is represented by a collection of opart records. The sum of units in each opart is equal to the number of units in the order. Each opart is a member of a product queue. If a machine group occurs more than once in the schedule of a product type, there may be more than one opart of the same order in the product queue of that machine group.

Among the attributes of an opart record are the following integers: The order number, ono, the product type, the step, the number of units waiting, nw, and the number of units in processing, np. The flow of units in the system is effected by counting up and down the attributes nw and np of opart records.

An opart record is generated at the time when the first batch of units of an order arrive at a machine group. It is entered at the end of the corresponding product queue. The opart will remain member of this queue till the last unit has entered processing. It will drop out of the system when the last unit has finished processing. A Boolean attribute last is needed to specify whether a given opart contains the last units of the order involved in this step.

At a given time the units of an order may be distributed on several machine groups. There will be an opart record for each
CHAPTER 13.

A WORKED EXAMPLE

The following example is given in terms of SIMULA. It demonstrates a technique of aggregating individual items into groups of items in order to increase program efficiency. The system described is a simplification of an actual case study carried out at the Norwegian Computing Center, Oslo.

Given a job shop consisting of machine groups, each containing a given number of identical machines in parallel. The system will be described from a machine point of view, i.e. the products flowing through the system are represented by processes which are passive data records. The machines operate on the products by remote accessing.

The products consists of orders, each for a given number of product units of the same type. There is a fixed number of product types. For each type there is a unique routing and given processing times.

For each machine group (number mg) there is a set avail[mg] of idle machines and a set que[mg], which is a product queue common to the machines in this group. The products are processed one batch at a time. One batch consists of a given number of units, which must belong to the same order. The batch size depends on the product type and the machine group.

A product queue is regarded as a queue of orders. The queue discipline is essentially first-in-first-out, the position of an order in the queue being defined by the arrival of the first unit of that order. However, if there is less than an acceptable
of them. An opart process will reference the one at the next step through an `element` attribute "successor". An order is thus represented by a simple chain of opart records. The one at the head has no successor, the one at the tail has its attribute "last" equal to true. The chain "moves" through the system by growing new heads and dropping off tails.

```
que[1]       que[j]       que[k]
```

![Diagram of opart process and machine groups](image)

machine group i  machine group j  machine group k

**Fig. 3.**

Three consecutive steps in the schedule of products of a given type. A product queue consists of oparts (circles) connected by vertical lines. Oparts belonging to the same order are connected by horizontal lines. Machines are represented by squares. A dotted line between an opart and a machine indicates a batch of units in processing. When the batch of the third opart in `que[j]` is finished, a new opart receiving this batch will be generated and included in `que[k]`.

The following piece of program is part of the head of a SIMULA block describing the above system. A machine activity is given. For clarity only statements essential for the behaviour of the model are shown. The program is not complete.
set array que, avail [1:nmg]; integer U;

integer procedure nextm (type, step); integer type, step;....;

real procedure ptime (type, step); integer type, step;....;

integer procedure bsize (type, mg); integer type, mg;....;

activity opart (ono, type, step, nw, np, last, successor);

integer ono, type, step, nw, np;
Boolean last; element successor;

activity machine (mg); integer mg;

begin integer batch, next; Boolean B; element X;

serve: X := head (que[mg]);

for X := suc (X) while exist (X) do

inspect X when opart do

begin batch := bsize (type, mg);

if nw < batch then begin

if last then batch := nw else go to no end;

nw := nw - batch; np := np + batch;

if last ∧ nw = 0 then remove (X);

activate first (avail[mg]);

hold (batch ∗ ptime (type, step) ∗ uniform (0.9, 1.1, U));

np := np - batch; B := last ∧ nw + np = 0;

next := nextm (type, step);

inspect successor when opart do

begin nw := nw + batch; last := B end

otherwise begin successor :=

new opart (ono, type, step + 1, batch, 0, B, none);

include (successor, que [next]) end;

activate first (avail [next]);

go to serve;

no; end;

wait (avail [mg]); remove (current); go to serve end;
Comments.

Line 1. The sets will contain oparts and idle machines respectively. The variable U defines a pseudo-random number stream (line 19).

Lines 2-4. The functions "nextm" and "ptime" specify the next machine group and the current processing time for a given product type and step in the schedule. "bsize" determines the batch size, given the product type and machine group number. The three functions are left unspecified.

Lines 5-7. The meanings of the attributes of opart processes have been explained above. The activity body is a dummy statement: an opart process is a data record with no associated actions.

Line 8. The machine activity extends to and includes line 30. The parameter mg is the machine group number. Machines belonging to the same group are completely similar.

Line 9. "batch" is the size of the current batch of units, "next" is the number of the next machine group for the units currently being processed, the meaning of "B" is explained below (line 20), and "X" is used for scanning.

Line 10. Prepare for scanning the appropriate product queue.

Line 11. Scan. The controlled statement is itself a connection statement (lines 12-29).

Line 12. There is only one connection branch (lines 12-29). Since a product queue contains only opart records connection must become effective. The attributes of the connected opart are accessible inside the connection block.
Line 13. Compute the standard batch size.

Lines 14, 15. A smaller batch is only accepted if the opart is at the tail end of the chain. In this case "nw" is nonzero (cf. line 17), and the units are the last ones of the order. Otherwise the next opart is tried.

Line 16. "batch" units are transferred from the waiting state to the in-processing state by reducing nw and increasing np.

Line 17. The opart is removed from the product queue when processing has started on the last units of the order.

Line 18. The current machine has found an acceptable batch of units, and has updated the product queue. There may be enough units left for another batch, therefore the next available machine in this group (mg) is activated. If there is no idle machine, the set avail[mg] is empty and the statement has no effect. See also lines 27 and 30.

Lines 19. The expected processing time is proportional to the number of units in the batch. The actual processing time is uniformly distributed in the interval ± 10% around the expected value. The sequence of pseudo-random drawings is determined by the initial value of the variable U.

Line 20. Processing is finished; np is reduced. The Boolean variable B gets the value true if and only if the last units of an order have now been processed. In that case the connected opart should drop off the chain at this system time (see comments to line 28). It follows that B is always the correct (next) value of the attribute "last" of the succeeding opart (lines 23, 25).
Line 21. Compute the number of the machine group to receive
the current batch of units.

Line 22. The **element** attribute "successor" is inspected.
The connection statement, lines 22-26, has two
branches.

Line 23. This is a connection block executed if "successor"
refers to an opart. The latter is a member of
the product queue of the next machine group. It
receives the processed batch of units, which are
entered in the waiting state. The attribute "last"
is updated. Notice that the attributes referenced
in this inner connection block are those belonging
to the successor to the opart connected outside (X).

Lines 24, 25. If the connected opart (X) is at the head of the
chain the value of "successor" is assumed equal
to **none**, and the **otherwise** branch is taken. A
new opart is generated, and a reference to it is
stored in "successor". The new opart has the
same "ono" and "type" as the old one, and its
"step" is one greater. It has "batch" units in
the waiting state and none in processing. Its
attribute "last" is equal to "B". Since the new
opart has become the head of the chain, its
"successor" should be equal to **none**. Notice that
the initial value of "last" may well be **true**, e.g. if the order contains a single unit.

Line 26. The new opart is included at the end of the product
queue of the next machine group.

Line 27. The current machine has now transferred a batch
of units to the product queue of next machine group.
Therefore the first available machine (if any) of
that group is activated. If that machine finds an
acceptable batch it will activate the next machine
in the same group (line 18). This takes care of
the case that the batch transferred is larger than the standard batch size of the next machine group for this type of product.

The machine immediately returns to the beginning of its operation rule to look for another acceptable batch starting at the front end of the product queue. At this point, if B is true, the connected opart is empty of units and will not be referenced any more. We can regard it has having dropped off the chain. It is easy, however, to demonstrate that the opart will physically leave the system, i.e. that its reference count is reduced to zero. The possible stored references are:

1) The variable X and the connection pointer "opart" of this machine or another one of the same group. The go to statement leads out of the connection block, which deletes the connection pointer. X is given another value in line 10. Any other machine referencing this opart would have to be suspended in line 19, which is impossible since np is zero (cf. the second statement of line 16).

2) Set membership in que[mg]. The opart must have been removed from the queue (by this machine or another one) since "last" is true and nw is now zero (line 17).

3) The attribute "successor" of the opart preceding this one in the chain. The first opart of this order to enter the system has no predecessor. Provided that this first one drops out when it is empty, our conclusion follows by induction (see below).
The end of the connection block and of the statement controlled by the for clause in line 11.

If, after having searched the entire product queue, the machine has found no acceptable batch, it includes itself in the appropriate "avail" set and goes passive. Its local sequence control remains within the wait statement as long as the machine is in the passive state. When the machine is eventually activated (by another machine: line 27 or 18), it removes itself from the "avail" set and returns to scan the product queue. The "avail" sets are operated in the first in-first out fashion.

The mechanism for feeding orders into the system is not shown above. This is typically done by the Main Program or by one or more "arrival" processes, which generate a pattern of orders, either specified in detail by input data, or by random drawing according to given relative average frequencies of product types and order sizes.

An arrival pattern defined completely "at random" is likely to cause severely fluctuating product queues, if the load on the system is near the maximum. The following is a simple way of rearranging the input pattern such as to achieve a more uniform load. The algorithm is particularly effective if there are different "bottle-necks" for the different types of products.

31. activity arrival (type, mg1, pt);
32. integer type, mg1; real pt;
33. begin integer units;
34. loop: select (units, type); id := id + 1;
35. include (new opart (id, type, 1, units, 0, true,
none), queue[1]);
36. activate first (avail [mg1]);
37. hold (pt*units); go to loop end;
38. procedure select (n, type); value type; integer n, type;....
39. integer id;
Comments.

Line 31. There will be one "arrival" process for each product type. "mg1" is the number of the first machine group in the schedule of this type of product. "pt" is a stipulated "average processing time" per unit, chosen so as to obtain a wanted average throughput of units of this type (see line 37).

Line 34. The procedure "select" should choose the size, "units", of the next order of the given type, e.g. by random drawing or by searching a given arrival pattern for the next order of this type. "id" is a non-local integer variable used for numbering the orders consecutively.

Line 35. An order is entered by generating an opart record which contains all the units of the order. The units are initially in the waiting state. The order is filed into the appropriate product queue. The set membership is the only reference to the opart stored by the arrival process. Consequently this opart will leave the system when it becomes empty of units, as assumed earlier (line 28).

Line 36. A machine in the appropriate group is notified of the arrival of an order.

Line 37. The next order of the same type is scheduled to arrive after a waiting time proportional to the size of this order, which ensures a uniform load of units (of each type).

The "output" of units from the system can conveniently be arranged by routing all products to a dummy machine group at the end of the schedule. It contains one or more "terminal machines" (not shown here) which may perform observational functions such as recording the completion of orders.
The dynamic setup of the system is a separate task, since initially the Main Program is the only process present. The Main Program should generate (and activate) all processes which are "permanent" parts of the system, such as machines, arrival processes and observational processes. The system can be started empty of products, however, a "steady" state can be reached in a shorter time if orders (opart records) are generated and distributed over the product queues in suitable quantities.

The experimental results are obtained by observing and reporting the behaviour of the system. Three different classes of outputs can be distinguished.

1) **On-line reporting.** Quantities describing the current state of the system can be printed out, e.g. with regular system time intervals: lengths of product queues in terms of units waiting, the total number of units in the system, the number of idle machines in each group, etc. A more detailed on-line reporting may be required for program debugging.

2) **Accumulated machine statistics.** By observing the system over an extended period of system time averages, extrema, histograms, etc., can be formed. Quantities observed can be queue lengths, idle times, throughputs, and so on. The accumulation of data could be performed by the machine processes themselves.

**Example.** To accumulate a frequency histogram of the idle periods of different lengths for individual machines, insert the following statements on either side of the "wait" statement of line 30:

"tidle := time" and "histo" (T, H, time - tidle, 1)", where "tidle" is a local real variable, and T and H are arrays. T[i] are real numbers which partition observed idle periods (time - tidle) into classes according to their lengths, and H[i] are integers equal to the number of
occurrences in each class. The system procedure "histo"
which will increase H[i] by one (the last parameter), where
i is the smallest integer such that T[i] is greater than or
equal to the idle period "time - tide". T and H together
thus define a frequency histogram, where T[i] - T[i - 1] is
the width of the i'th column, and H[i] is the column length.

3) **Accumulated order statistics.** During the life time of an
opart record the "history" of an order at a given machine
group can be accumulated and recorded in attributes of the
opart. The following are examples of data which can be
found.

The arrival time of the first unit of the order at this
machine group is equal to the time at which the opart is
generated. The departure time of the last unit is equal to
the time at which the variable B gets the value **true**
(line 20 of a machine connecting the opart).

The sum of waiting times for every unit of the order in
this queue is equal to the integral with respect to system
time of the quantity nw (which is a step function of time).
The integral can be computed by the system procedure "accum".
The statements "nw := nw + batch" (lines 16 and 23) are
replaced by "accum (anw, tnw, nw, + batch)", where the **real**
variables anw and tnw are additional attributes of the
opart process, with initial values zero and "time"
respectively. The procedure will update nw and accumulate
the integral in anw. It is equivalent to the statements:

\[
anw := anw + nw \times (time - tnw); \quad tnw := time; \quad nw = nw + batch;
\]

It is worth noticing that arrival times, waiting times, etc.,
can not in general be found for individual units, unless
the units are treated as individuals in the program. Neither
can the maximum individual waiting time for units in an
order. The average waiting time, however, is equal to the
above time integral divided by the number of units in the
order.
The complete history of an order in the shop is the collection of data recorded in the different parts of the order. These data can be written out on an external storage medium at the end of the lifetime of each part. I.e. an output record could be written out before line 28, whenever B is true, containing items such as the order number, ono, the sum of waiting times, anw, the current system time, etc. When the simulation has been completed, the data records can be read back in, sorted according to order numbers, and processed to obtain information concerning the complete order, such as the total transit time, total waiting time etc.

The same information can be obtained by retaining the complete part chain in the system until the order is out of the shop. However, this requires more memory space. The chain can be retained by making the arrival process include the initial part in an auxiliary set, or by having a pointer from the part currently at the head of the chain back to the initial one. The part chain can be processed by the terminal machine. (The order is completely through the shop at the time when the attribute "last" of the part in the terminal product queue gets the value true.) In the former case the terminal machine should also remove the appropriate part from the auxiliary set, in order to get rid of the part chain.
## A. Alphabetic order.

<table>
<thead>
<tr>
<th>Name</th>
<th>Result Arithmetic</th>
<th>No.of.param</th>
<th>Report reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCUM</td>
<td>none</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>CANCEL</td>
<td>-</td>
<td>1</td>
<td>4.8</td>
</tr>
<tr>
<td>CARDINAL</td>
<td>integer</td>
<td>1</td>
<td>3.7</td>
</tr>
<tr>
<td>CLEAR</td>
<td>none</td>
<td>1</td>
<td>3.7</td>
</tr>
<tr>
<td>CURRENT</td>
<td>element</td>
<td>0</td>
<td>4.4</td>
</tr>
<tr>
<td>DISCRETE</td>
<td>integer</td>
<td>2</td>
<td>7.2</td>
</tr>
<tr>
<td>DRAW</td>
<td>Boolean</td>
<td>2</td>
<td>7.2</td>
</tr>
<tr>
<td>EMPTY</td>
<td>-</td>
<td>1</td>
<td>3.7</td>
</tr>
<tr>
<td>EQUAL</td>
<td>-</td>
<td>2</td>
<td>(no reference)</td>
</tr>
<tr>
<td>EvTIME</td>
<td>real</td>
<td>1</td>
<td>4.4</td>
</tr>
<tr>
<td>EXST</td>
<td>Boolean</td>
<td>1</td>
<td>3.7</td>
</tr>
<tr>
<td>FINISHED</td>
<td>-</td>
<td>1</td>
<td>4.4</td>
</tr>
<tr>
<td>FIRST</td>
<td>element</td>
<td>1</td>
<td>3.7</td>
</tr>
<tr>
<td>FOLLOW</td>
<td>none</td>
<td>2</td>
<td>3.7</td>
</tr>
<tr>
<td>HEAD</td>
<td>element</td>
<td>1</td>
<td>3.4.2</td>
</tr>
<tr>
<td>HISTD</td>
<td>integer</td>
<td>2</td>
<td>7.2</td>
</tr>
<tr>
<td>HISTO</td>
<td>none</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>OLD</td>
<td>-</td>
<td>1</td>
<td>4.3</td>
</tr>
<tr>
<td>IDLE</td>
<td>Boolean</td>
<td>1</td>
<td>4.4</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>none</td>
<td>2</td>
<td>3.7</td>
</tr>
<tr>
<td>LAST</td>
<td>element</td>
<td>1</td>
<td>3.7</td>
</tr>
<tr>
<td>LINEAR</td>
<td>real</td>
<td>3</td>
<td>7.2</td>
</tr>
<tr>
<td>MEMBER</td>
<td>element</td>
<td>2</td>
<td>3.7</td>
</tr>
<tr>
<td>NCEXP</td>
<td>real</td>
<td>2</td>
<td>7.2</td>
</tr>
<tr>
<td>NEXTEV</td>
<td>element</td>
<td>1</td>
<td>4.4</td>
</tr>
<tr>
<td>NORMAL</td>
<td>real</td>
<td>3</td>
<td>7.2</td>
</tr>
<tr>
<td>NUMBER</td>
<td>element</td>
<td>2</td>
<td>3.7</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>integer</td>
<td>1</td>
<td>3.7</td>
</tr>
<tr>
<td>PASSIVATE</td>
<td>none</td>
<td>0</td>
<td>4.2</td>
</tr>
<tr>
<td>POISSON</td>
<td>integer</td>
<td>2</td>
<td>7.2</td>
</tr>
<tr>
<td>PHCD</td>
<td>none</td>
<td>2</td>
<td>3.6</td>
</tr>
<tr>
<td>PRECEDE</td>
<td>-</td>
<td>2</td>
<td>3.7</td>
</tr>
<tr>
<td>PRED</td>
<td>element</td>
<td>1</td>
<td>3.4.2</td>
</tr>
<tr>
<td>PROC</td>
<td>-</td>
<td>1</td>
<td>3.4.1</td>
</tr>
<tr>
<td>PSNORM</td>
<td>real</td>
<td>4</td>
<td>7.2</td>
</tr>
<tr>
<td>RANDINT</td>
<td>integer</td>
<td>3</td>
<td>7.2</td>
</tr>
<tr>
<td>REMOVE</td>
<td>none</td>
<td>1</td>
<td>3.6</td>
</tr>
<tr>
<td>SAME</td>
<td>Boolean</td>
<td>2</td>
<td>3.5</td>
</tr>
<tr>
<td>SIMILAR</td>
<td>-</td>
<td>2</td>
<td>3.5</td>
</tr>
<tr>
<td>SUC</td>
<td>element</td>
<td>1</td>
<td>3.4.2</td>
</tr>
<tr>
<td>SUCCESSOR</td>
<td>-</td>
<td>2</td>
<td>3.7</td>
</tr>
<tr>
<td>TERMINATE</td>
<td>none</td>
<td>1</td>
<td>4.2</td>
</tr>
<tr>
<td>TIME</td>
<td>real</td>
<td>0</td>
<td>4.4</td>
</tr>
<tr>
<td>TRNSFER</td>
<td>none</td>
<td>2</td>
<td>3.7</td>
</tr>
<tr>
<td>UNIFORM</td>
<td>real</td>
<td>3</td>
<td>7.2</td>
</tr>
<tr>
<td>WAIT</td>
<td>none</td>
<td>1</td>
<td>4.2</td>
</tr>
</tbody>
</table>
### B. Arithmetic order

#### 1. No arithmetic

<table>
<thead>
<tr>
<th>Name</th>
<th>N.of p.</th>
<th>Param. types</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCUM</td>
<td>4</td>
<td>E</td>
</tr>
<tr>
<td>CANCEL</td>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td>CLEAR</td>
<td>1</td>
<td>E,E</td>
</tr>
<tr>
<td>FOLLOW</td>
<td>2</td>
<td>A,A,R,R</td>
</tr>
<tr>
<td>HISTO</td>
<td>4</td>
<td>R</td>
</tr>
<tr>
<td>HOLD</td>
<td>1</td>
<td>E,S</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>PASSIVATE</td>
<td>20</td>
<td>E,E</td>
</tr>
<tr>
<td>PRCD</td>
<td>2</td>
<td>E,E</td>
</tr>
<tr>
<td>PRECEDE</td>
<td>2</td>
<td>E</td>
</tr>
<tr>
<td>REMOVE</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>TERMINATE</td>
<td>1</td>
<td>E,S</td>
</tr>
<tr>
<td>TRANSFER</td>
<td>2</td>
<td>S</td>
</tr>
<tr>
<td>WAIT</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

#### 2. Integer

<table>
<thead>
<tr>
<th>Name</th>
<th>N.of p.</th>
<th>Param. types</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARDINAL</td>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td>DISCRETE</td>
<td>2</td>
<td>RA,I</td>
</tr>
<tr>
<td>HISTD</td>
<td>2</td>
<td>RA,I</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td>POISSON</td>
<td>2</td>
<td>R,I</td>
</tr>
<tr>
<td>RANDINT</td>
<td>3</td>
<td>I,I,I</td>
</tr>
</tbody>
</table>

#### 3. Real

<table>
<thead>
<tr>
<th>Name</th>
<th>N.of p.</th>
<th>Param. types</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVTIME</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>LINEAR</td>
<td>3</td>
<td>RA,RA,I</td>
</tr>
<tr>
<td>NEGEXP</td>
<td>2</td>
<td>R,I</td>
</tr>
<tr>
<td>NORMAL</td>
<td>3</td>
<td>R,R,I</td>
</tr>
<tr>
<td>PSNORM</td>
<td>4</td>
<td>R,R,I,I</td>
</tr>
<tr>
<td>TIME</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>UNIFORM</td>
<td>3</td>
<td>R,R,I</td>
</tr>
</tbody>
</table>

#### 4. Boolean

<table>
<thead>
<tr>
<th>Name</th>
<th>N.of p.</th>
<th>Param. types</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAW</td>
<td>2</td>
<td>R,I</td>
</tr>
<tr>
<td>EMPTY</td>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td>EQUAL</td>
<td>2</td>
<td>E,E</td>
</tr>
<tr>
<td>EXIST</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>FINISHED</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>IDLE</td>
<td>1</td>
<td>E,E</td>
</tr>
<tr>
<td>SAME</td>
<td>2</td>
<td>E,E</td>
</tr>
<tr>
<td>SIMILAR</td>
<td>2</td>
<td>E,E</td>
</tr>
<tr>
<td>Name</td>
<td>N.of p.</td>
<td>Param. types</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>--------------</td>
</tr>
<tr>
<td>CURRENT</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>FIRST</td>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td>HEAD</td>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td>LAST</td>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td>MEMBER</td>
<td>2</td>
<td>E,S</td>
</tr>
<tr>
<td>NEXTEV</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>NUMBER</td>
<td>2</td>
<td>I,S</td>
</tr>
<tr>
<td>PRED</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>PROC</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>SUC</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>SUCCESSOR</td>
<td>2</td>
<td>I,E</td>
</tr>
</tbody>
</table>