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 refe-
renceable, 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
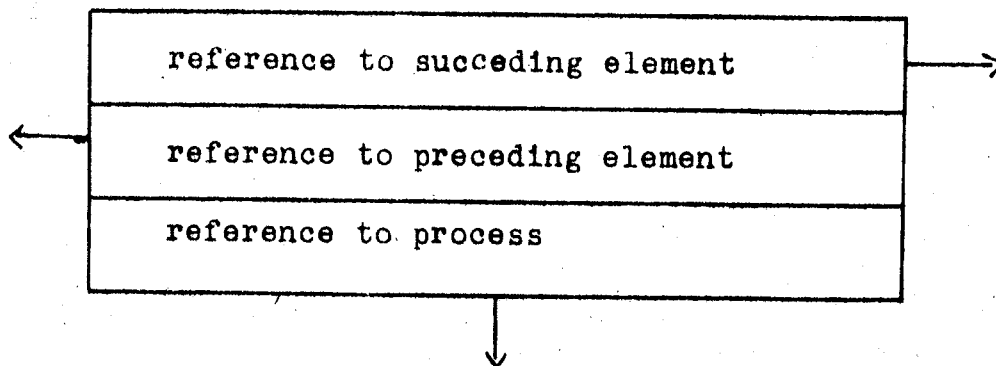by fig. 1.


```
┌─────────────────────────────────────┐
│  reference to succeding element      │──────→
├─────────────────────────────────────┤
←─│  reference to preceding element      │
├─────────────────────────────────────┤
│  reference to process                │
└─────────────────────────────────────┘
                  │
                  ↓
```

Fig. 1

The references to the succeding 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.
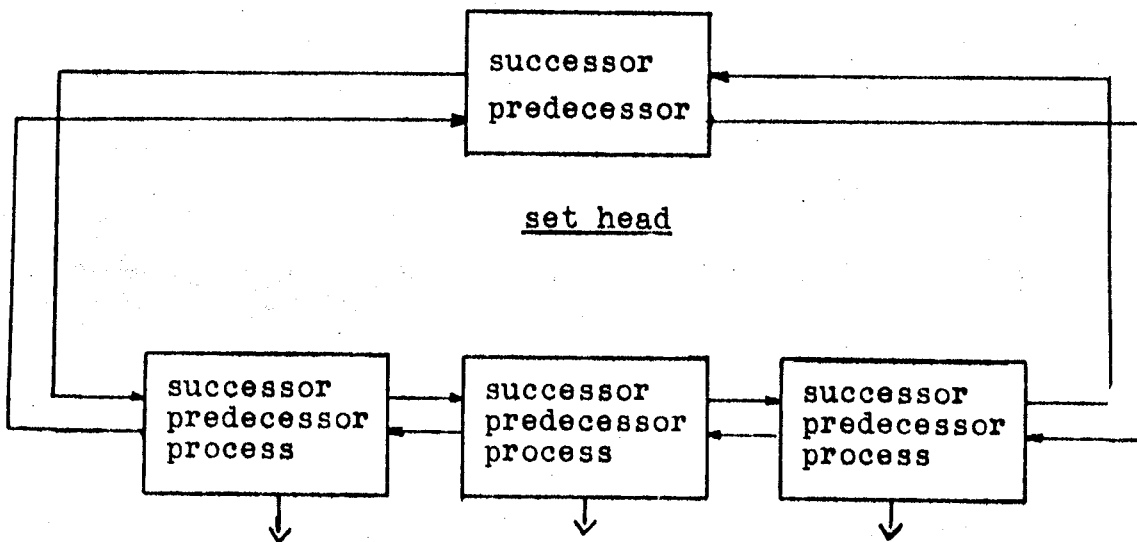
```
              ┌──────────────────┐
              │    successor     │
              │   predecessor    │
              └──────────────────┘
                   set head

┌──────────────┐  ┌──────────────┐  ┌──────────────┐
│  successor   │  │  successor   │  │  successor   │
│ predecessor  │  │ predecessor  │  │ predecessor  │
│  process     │  │  process     │  │  process     │
└──────────────┘  └──────────────┘  └──────────────┘
```

Fig. 2

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 prede-
cessor. 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), pro-
cess 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 (redhaired, greeneyed, thumbs);
       Boolean redhaired, 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. suc(X)   denotes the successor of X. If X has no SM,
   suc(X) is <u>none</u>. If X has a SM, suc(X) is a unique element
   having itself a SM.

3. pred(X)   denotes the predecessor of X. If X has no SM,
   pred(X) is <u>none</u>. If X has a SM pred(X) is a unique
   element having itself a SM.

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

3.5 <u>Boolean expressions</u>.

The following are basic <u>Boolean</u> expressions applying to ele-
ments.

1. "X = Y" is <u>true</u> if either X and Y denote the same element
   or both values are <u>none</u>.

2. "X ≠ Y" is the negation of "X = Y".

3. same(X,Y)   is <u>true</u> if either X and Y reference the same
   process or neither has a PA.
   "X = Y" implies same(X,Y).

4. similar(X,Y)   is <u>true</u> if either X and Y reference processes
   of the same class or neither has a PA.
   same(X,Y) implies similar(X,Y).

Examples.

"X ≠ X" is <u>true</u> if X is a generative expression.
same(X, proc(X)) is <u>true</u> if X evaluates to the same element both
times.
similar(<u>new</u> A, <u>new</u> A) is <u>true</u>.
same(head(S), <u>none</u>) is <u>true</u>, but "head(S) = <u>none</u>" is <u>false</u>.

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. prcd(X,Y)

   If X has a SM, say, pred(X) = Z, Z $\neq$ 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 prcd(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 $\neq$ none, then suc(Y) and pred(Y) become none, suc(Z)
   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.

$$pred(Z,X) \; ; \; Y: = X \; ; \; remove(X);$$

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 proce-
dures, although they are all expressible in terms of basic
SIMULA concepts.

element procedures

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

```
element procedure successor(n,X); value n, X;
integer n; element X;
begin integer i;
```

```
i: = 0;
for i: = i+1 while  i < abs(n) ∧ ¬ same (X, none) do
X: = if  n > 0 then suc(X)  else pred (X);
successor: = X; end;
```

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

5. member (X,S) denotes an element of S with the same PA as
   that of X. 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.

```
element procedure member(X,S); value X,S;
element X; set S;
begin element Y; member := none;
for Y := head(S), suc(Y) while Y ≠ head(S) do
if same(X,Y) then begin member := Y; go to
fin end;
fin:  end;
```

Boolean procedures

1. exist(X)   is equivalent to ¬ same(X, none); i.e. the
   value is true if X has a process aspect.

2. empty(S)   is equivalent to ¬ exist(first(S))

integer procedures

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

```
integer procedure ordinal(X); value X; element X;
begin integer i; i := 0; X := suc(X);
for X := pred(X) while exist(X) do i := 1+1;
ordinal := i end;
```

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) evalu-
ates to another element each time.

2. Set subtraction.

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

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.

for X: = first(T), suc(X)while exist(X) do remove(member(X, S));