Norwegian Computing Center
Forskningsveien 1B
Oslo 3
Norway.

Publication no. S-44
February 1973

SIMULA - its features and prospects

by

G. M. Birtwistle

A lecture delivered at the B.C.S.
Conference on "High level languages",
York, October 1972.

SIMULA - its features and prospects

G. M. Birtwistle, Norwegian Computing Center (NCC)

In this lecture, I shall concentrate upon giving you a view of SIMULA's structuring properties as time does not permit me to go into details. A full account is given in [9]. I shall briefly outline the history of SIMULA, talk at some length on the most important features, and then look briefly at the prospects for its future.

History

SIMULA [2,8] is based upon ALGOL 60. The authors found that ALGOL 60's block structure, suitably extended, was an ideal foundation for mirroring complex situations involving many components. The first SIMULA [1], now known as SIMULA I, was a simulation language implemented on the UNIVAC 1107. Shortly afterwards, UNIVAC took over the maintenance of the compiler, and Dahl and Nygaard, now joined by Myhrhaug, began the development of its successor, the new SIMULA about which I am to talk. It has further developed the co-routine concept of SIMULA I and extended it to make the language more general purpose. In addition, SIMULA contains complete alphanumeric string handling facilities and well defined I/O. But I have no time to go into these.

At the moment, SIMULA is available on the CD 3000/6000 series, the UNIVAC 1100 series (EXEC 8) and the IBM 360/370 series (under OS) - amongst others. Comparisons of SIMULA's compilation and execution speeds are given in [11,12].

Language development is controlled by the SIMULA STANDARDS GROUP which decides upon amendments and extensions to the Common Base [2].

A SIMULA application package - CLASS DRAUGHTING

We introduce the main features of SIMULA by outlining an application package written in SIMULA. This package is fully described, with listings, in Kjeldaas [13].

The package defines the geometrical concepts of points, lines, figures, etc. and enables the description of geometrical constructions to be made in a fairly natural notation. Notice that the way we picture the concepts of the problem area is very closely mirrored by the SIMULA descriptions.

A geometrical point is completely specified by two cartesian co-ordinates (x,y). But it may have other "attributes" associated with it too - for example, its distance "r" from the origin, the angle, "theta" it makes with the positive x-axis. In SIMULA, we describe such a concept in a "class declaration". The class declaration for points could be:

```
CLASS POINT(X,Y); REAL X,Y;
BEGIN REAL R, THETA;
      REF(POINT) PROCEDURE ROTATED .....;
      BOOLEAN PROCEDURE EQUALS .....;
      REAL PROCEDURE DISTANCE .....;
      R := SQRT(X↑2+Y↑2);
      THETA := ARCTAN2(X,Y);
END **** POINT **** ;
```

where ARCTAN2 is some suitable procedure which returns the angle in degrees (0→360) made by the point (x,y) to the positive x-axis.

The declaration has a name (POINT) and parameters; within the body of the class, further attributes are defined.

ROTATED(P,N) will generate a new point by rotating this one about the point P through N degrees,

EQUALS(P) will check if this point and P represent the same geometrical point,

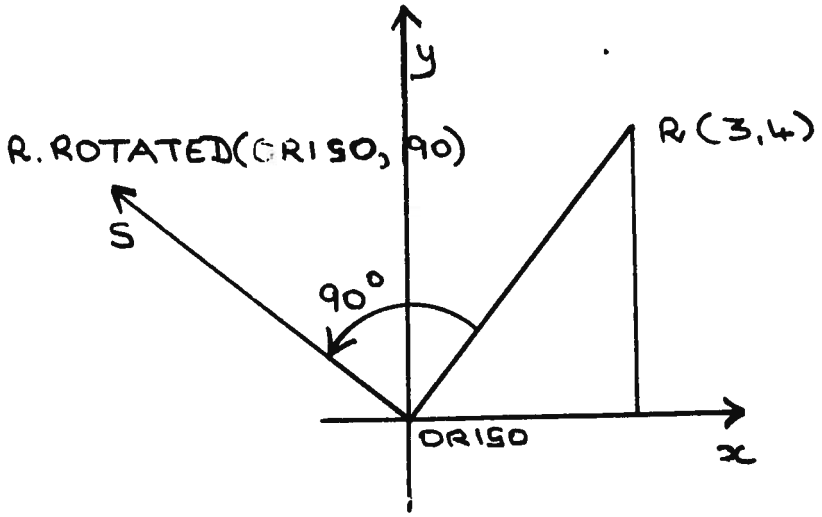
DISTANCE(P) returns the distance between this point and the point P.

ROTATE, EQUALS and DISTANCE may be viewed as operators working on the local co-ordinates x,y. In addition, R and THETA are not only locally defined, but initialised by the actions of the class body. Each time a point is created, R and THETA are evaluated from the actual parameter values for x and y. This feature - associating actions with records - is unique to SIMULA. Here, in a simple case, it shows how we can reduce the number of parameters. Further uses will be noted later.

We create and name objects of the class point in the following way:

```
REF(POINT) R,S,ORIGO;
ORIGO :- NEW POINT(0,0);
R     :- NEW POINT(3,4);
S     :- R.ROTATED(ORIGO,90);
```

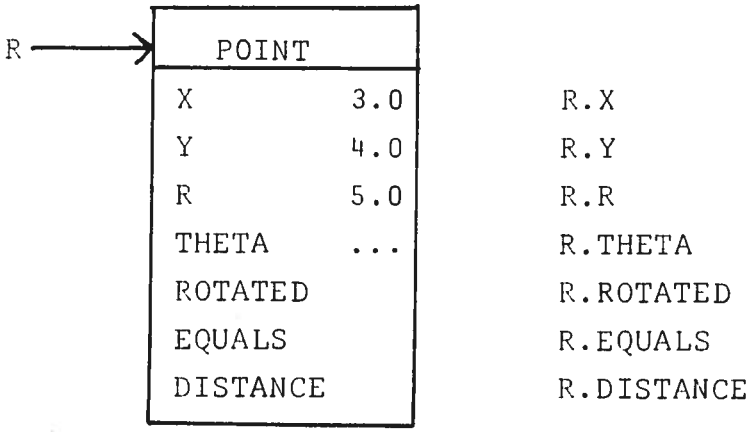
This coding creates three variables R,S, ORIGO which will be used to reference point objects, and then creates representations of the origin, the point (3,4) and the point which is the point (3,4) rotated about the origin through 90^{o}.



Attributes of a "named" object are accessed by

<object name>.<attribute name>

e.g.



so that R.ROTATED is a call on the procedure ROTATED embedded in the object currently referenced by R.

When we declare reference variables in SIMULA, they are given a "qualification", e.g. POINT in REF(POINT). Reference variables may only reference objects "in" that class. By using a qualification we may (nearly always) ensure at compile time that a reference variable is guaranteed at run time to reference either NONE (no object at all) or an object whose data structure is known - i.e. the validity of data accessing can be checked at compile time. The exceptions are due to hierarchical class definitions which allow a reference variable to reference objects of subclasses of their qualification. See [9, Chapter 4.2] for a full explanation.

The second declaration we give describes the concept of a chord:

```
CLASS CHORD(P,Q); REF(POINT) P,Q;
BEGIN REAL LENGTH;
      IF P==NONE OR Q==NONE OR P.EQUALS(Q)
         THEN ERROR("ILLEGAL CHORD DEFINITION")
         ELSE LENGTH := P.DISTANCE(Q);
END **** CHORD **** ;
```

Here we define a chord in terms of its end points. Notice that whenever a chord object is created, we check that it is properly defined, i.e. its end points exist and are distinct. In a more general context then, the class actions may be used to check the validity of the data associated with a record upon its creation.

The code to represent the chord between R and S is just:

```
REF(CHORD)C;
C :- NEW CHORD(R,S);
```

Prefixed levels

Often when working in problem areas, we want to build up libraries of inter-related concepts. This can be done in SIMULA by a so-called "prefix class". In this case, we might develop:

```
CLASS DRAUGHTING;
BEGIN CLASS POINT ..............;
      CLASS CHORD ..............;
      CLASS LINE ...............;
      CLASS POLYGON ............;
      CLASS FIGURE .............;

      REF(LINE) XAXIS, YAXIS;
      REF(POINT) ORIGO;

      XAXIS :- NEW LINE ........;
      YAXIS :- NEW LINE ........;
      ORIGO :- NEW POINT(0,0);
END **** DRAUGHTING **** ;
```

DRAUGHTING is the name of a package containing many handy geometrical building blocks. On calling the class DRAUGHTING, all the concepts it contains become available to the programmer for direct use. The DRAUGHTING example also shows how the actions may be used to completely set the environment for the user - in this case the axes and origin are generated for him.

We now give a small program to draw a simple figure. The coding uses the concepts contained in Kjeldaas [13], and although the concepts of line, polygon and figure have not been formalised here, the program is hopefully readable enough.

```
DRAUGHTING
    BEGIN
        COMMENT by prefixing a user block all the concepts
        in DRAUGHTING become available to the user;
        REF(POINT) R,S; REF(POLYGON) P;
        REF(FIGURE) F; INTEGER I;

        COMMENT generate the points (10,0), (11,0);
        R :- NEW POINT(10,0);
        S :- NEW POINT(11,0);

        COMMENT generate the polygon P which contains 6
        points - S, and the 60°, 120°, ..., 300° rotations
        of S about R;
        P :- NEW POLYGON;
        S.INTO(P);
        FOR I := 1 STEP 1 UNTIL 5 DO
          S.ROTATED(R,60*I).INTO(P);

        COMMENT generate the figure F which will hold
        6 polygons - P and the 60°, 120°, ..., 300°
        rotations of P about the origin;
        F :- NEW FIGURE;
        P.INTO(F);
        FOR I := 1 STEP 1 UNTIL 5 DO
          P.ROTATED(ORIGO,60*I).INTO(F);

        COMMENT sketch the figure;
        F.PLOT;
    END
```

This program shows how SIMULA can be used to generate and use packages for a prescribed problem area. DRAUGHTING forms a substantial kernel from which the user can model a geometrical construction in geometrical terms and build up the final figure in a natural way. One of the authors of SIMULA, Nygaard, has become intensely interested in the use of SIMULA as a tool for system description [9,10].

DRAUGHTING is an example of a prefix class - a collection of interrelated concepts worked out once and for all, yet extendible if need be (see next section). By writing

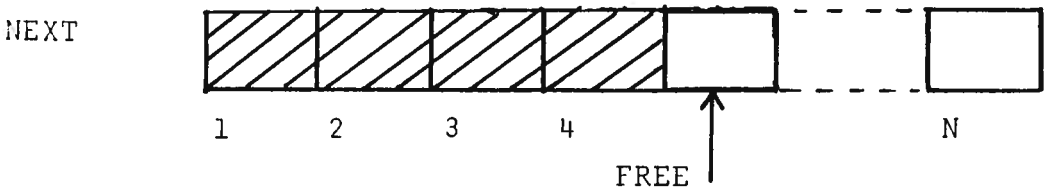
DRAUGHTING BEGIN

the whole armoury is brought into use. A prefix class then forms a conceptual platform which can be used to reduce the gap between the available software, SIMULA, and the problem at hand. The gap can, in general, be further reduced by building layer upon layer. Each layer is more and more user oriented, and further extends the previously defined concepts towards the specific problem area. We start with primitive ideas and gradually mould them into more useful structures. SIMULA is thus oriented towards bottom-up programming - writing a broad base which can be of use in several areas - and pre-compilation.

"Before your very eyes" - examples of prefix layers

We now use SIMULA's "level" technique to build a list-processing scheme for a FORTRAN/ALGOL 60 environment. The fundamental problem with list-processing is that of storage (de)allocation and it is that problem we tackle first.

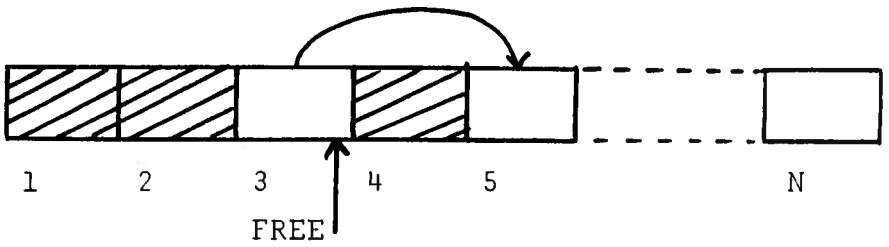
Suppose we put a "mask" on our store which allows for up to N list members. The mask is INTEGER ARRAY NEXT(1:N). Associated with the array is a pointer FREE which references the next available slot. Let us assume that the first four words of NEXT are in use.



To allocate and deallocate storage, we write two procedures, DELETE and LOCATE.

```
PROCEDURE DELETE(K); INTEGER K;
   IF K < 1 OR K > N THEN ERROR ELSE
      BEGIN
         NEXT(K) := FREE; FREE := K;
      END *** DELETE *** ;
```

A call DELETE(3) would produce:



To locate the free space and update the pointer FREE, we write:

```
INTEGER PROCEDURE LOCATE;
   IF FREE = 0 THEN ERROR ELSE
      BEGIN
         LOCATE := FREE;
         FREE := NEXT (FREE);
      END ***LOCATE*** ;
```

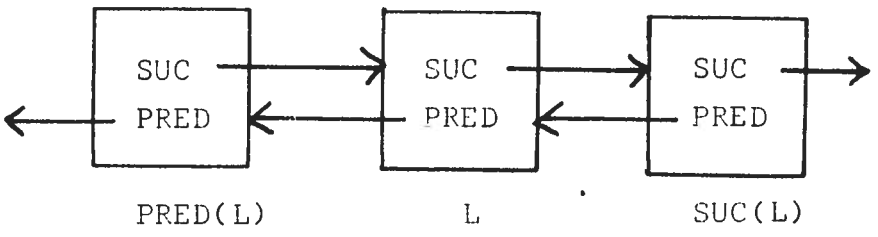
We collect these ideas together into a layer as follows:

```
CLASS STORAGE ALLOCATION (N); INTEGER N;
   BEGIN
       INTEGER ARRAY NEXT (1:N);
       INTEGER FREE;
       INTEGER PROCEDURE LOCATE ...;
       PROCEDURE DELETE ..........;
       FOR FREE := 1 STEP 1 UNTIL N-1 DO
           NEXT (FREE) := FREE + 1;
       FREE := 1;
   END ****STORAGE ALLOCATION****  ;
```

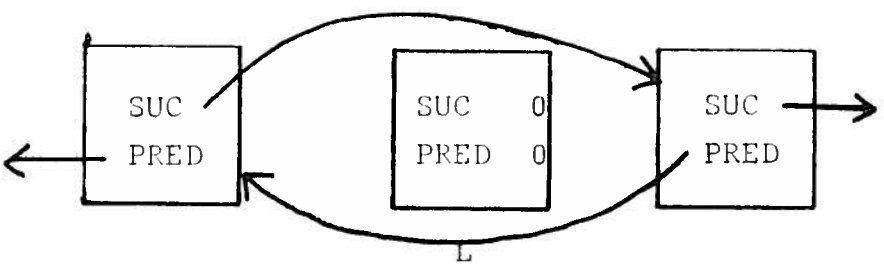
STORAGE ALLOCATION allows for an arbitrary number of list members (N > 0). Notice that we correctly initialise the array elements of NEXT to point to their successors (NEXT(N) is initialised by default to 0 in SIMULA), and FREE to 1.

Now that the storage (de)allocation problem has been described, we build a two way list processing package upon the STORAGE ALLOCATION platform. In two way list processing, each element has a reference to its predecessor and successor. To remove the list member referenced by the index L, we must "blank out" its own successor and predecessor pointers and update their references to it by making them jump around L.

before



after



The coding is:

```
PROCEDURE OUT(L); INTEGER L;
BEGIN
    IF L < 1 OR L > N THEN ERROR;
    PRED(SUC(L)) := PRED(L);
    SUC(PRED(L)) := SUC(L);
    SUC(L) := PRED(L) := 0;
    DELETE(L);
END **** OUT **** ;
```

where DELETE is a call on the procedure defined at the previous level, STORAGE ALLOCATION.

Similarly, a procedure to place an item into a list preceding a given item is:

```
PROCEDURE PRECEDE(L); INTEGER L;
BEGIN  INTEGER WHERE;
    IF L < 1 OR L > N THEN ERROR;
    WHERE := LOCATE;
    SUC(WHERE) := L;
    PRED(WHERE) := PRED(L);
    SUC(PRED(L)) := PRED(L) := WHERE;
END ****PRECEDE****;
```

In a similar way, we can define further procedures INTO and FOLLOW and complete the description of the second layer.

```
STORAGE ALLOCATION CLASS TWO WAY LIST;
    BEGIN
        INTEGER ARRAY SUC,PRED (0:N);
        PROCEDURE OUT ........;
        PROCEDURE PRECEDE ....;
        PROCEDURE INTO .......;
        PROCEDURE FOLLOW .....;
        SUC(0) := PRED(0) := 0;
    END
```

Notice that we build the first layer STORAGE ALLOCATION into this layer by simply prefixing the class declaration of TWO WAY LIST. TWO WAY LIST parallels SLIP in that we have implicitly a list HEAD, index 0, whose SUC and PRED are set by the class actions to initially reference itself.

Then as users who wish to read a list of at most one thousand members, rank it and count the number of incidences of items, we simply need to write two procedures and a two line program. The concepts brought down by TWO WAY LIST do the rest.

```
TWO WAY LIST (1000)
     BEGIN
          REAL ARRAY VAL (1:N);
          INTEGER ARRAY INCIDENCES (1:N);
          PROCEDURE RANK ..............;
          PROCEDURE WRITELIST ..........;

          WHILE ¬ LASTITEM DO RANK(INREAL);
          WRITELIST;
     END
```

In this example I have tried to show how a user can cut down the amount of work he has to do by building upon what is already there. We started with SIMULA and built a system much nearer that of the user. In general, anybody can take any level and use it for his particular problem. For example, we do not have to start from scratch if we are interested in one-way list processing or tree sorts. The storage problem has been solved, and we write

```
STORAGE ALLOCATION CLASS ONE WAY LIST .....;  , or
STORAGE ALLOCATION CLASS TREE SORT ........;
```

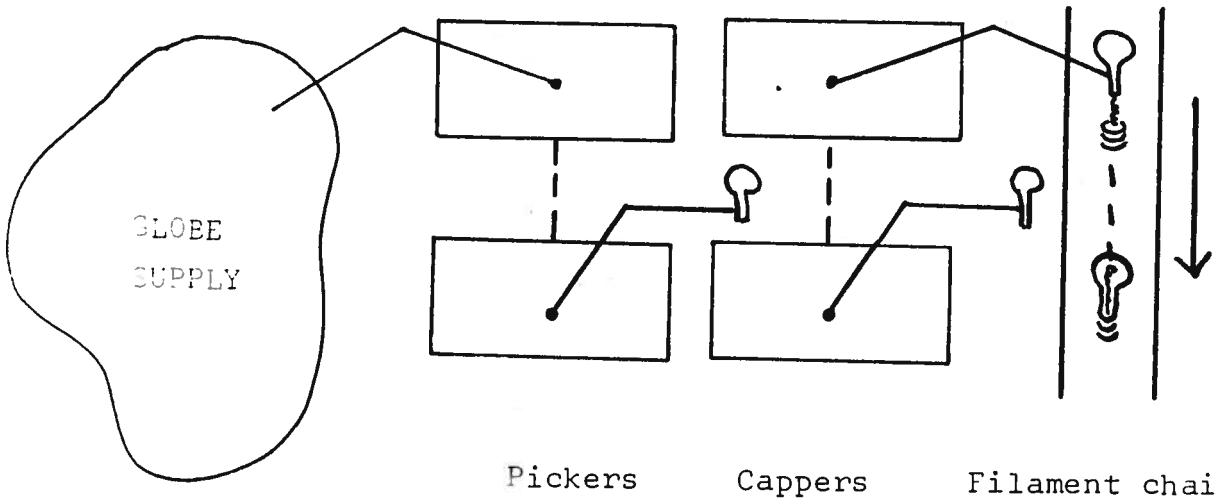
Again, if we wish to rank strings using two way lists, we would use TWO WAY LIST to prefix our program block where we supply new definitions of RANK and WRITELIST.

Objects

I would now like to turn to the second key concept of SIMULA - the object. We have already seen examples of objects (POINTS and CHORDS). Such objects have parameters, local data, local operators (procedures) and actions. The idea is generalised in SIMULA so that these actions may be obeyed in stages, and not all at once when the object is generated. This is useful in simulation where each entity in the real world may be modelled by a SIMULA object and its real time active and passive (from the point of view of not materially affecting the state of the system) phases may be modelled by having the corresponding object in action or being frozen.

The simulation features of SIMULA are presented in the form of two system defined prefix layers. CLASS SIMSET serves the same function as TWO WAY LIST using references instead of arrays. Further, since garbage collection is automatic and implicit, such a class as STORAGE ALLOCATION is completely unnecessary. SIMSET is then used as prefix to CLASS SIMULATION which contains the notions of simulation time, processes which can be scheduled in an event list, and scheduling statements. These concepts become alive in SIMULA programs by prefixing a user block with SIMULATION. The user merely has to remember to prefix all his process declarations by the system prefix PROCESS.

Simulation of a filament capping line



Problem description:

Uncapped bulbs move on a conveyor belt which moves for 2T time units, then still for 6T. Opposite each stationary position is a picker-capper pair which put bulbs on uncapped filaments.

Capping takes 3T time units, then the "capper" gets a new bulb from its partner "picker" if the picker has one available. Otherwise it waits until its partner has one.

A picker takes 7-11 time units to retrieve a bulb, orient it, and move to the transfer position. The transfer of a bulb from picker to capper is instantaneous.

Simulation objective:

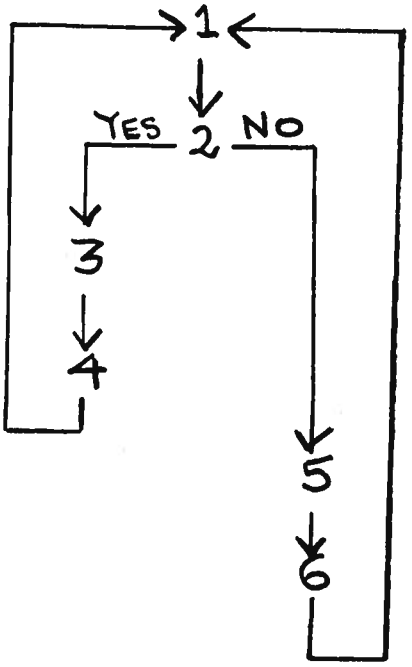
Determine how many filaments go uncapped with N picker-capper machines. Eventually, which N value is optimal.

In SIMULA, we can program by having a moving filament chain and a number of pickers (all alike) and cappers (all alike). We work on these three concepts one at a time.

For the picker, we can outline

ATTRIBUTES: Status; i.e. ready with globe, or not ready

ACTIONS:



1. Fetch a globe
2. Is the capper waiting for a globe?
3. Transfer the globe
4. Alert the capper
5. Wait
6. Transfer the globe

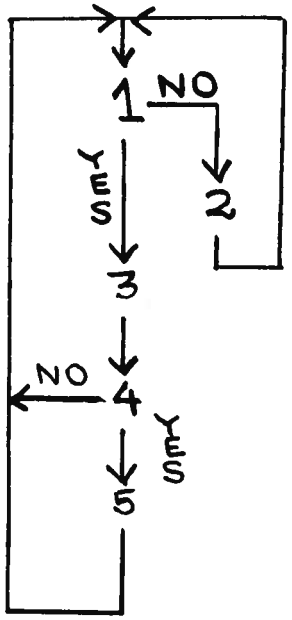
A semi formal description would be

```
PROCESS CLASS PICKER;
BEGIN REF (CAPPER) PARTNER;
      STATUS ...;
      ...

NEXTBULB:
         HOLD(uniform drawing 7 and 11);
         set status to "ready";
         IF partner is waiting for a globe
         THEN BEGIN transfer the globe; ACTIVATE PARTNER; END
         ELSE BEGIN PASSIVATE; transfer the globe; END;
         reset status to "not ready";
         GO TO NEXTBULB;
END **** PICKER **** ;
```

Similarly, for a capper we get:

ATTRIBUTES: Status: i.e. ready to cap, or waiting for a globe

ACTIONS:



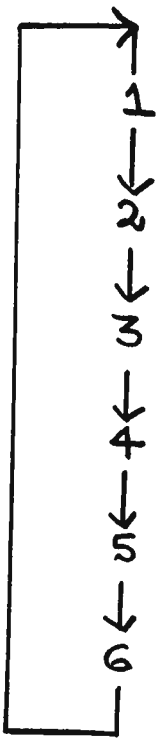
1. Are conditions for capping fulfilled?
2. Wait
3. Cap the bulb
4. Is the picker ready with a globe?
5. Activate the picker

```
PROCESS CLASS CAPPER (I); INTEGER I;
BEGIN   REF(PICKER) PARTNER;
        STATUS ...;
        ...

NEXTTRY:

        WHILE conditions for capping are not fulfilled
              DO PASSIVATE;
        HOLD (time to cap bulb);
        details of the capping;
        IF partner is ready with a globe THEN
           ACTIVATE PARTNER;
        GO TO NEXTTRY;
END **** CAPPER **** ;
```

And for the filament chain:

ATTRIBUTES: OK to cap; a signal to say whether the chain is stationary or moving.

ACTIONS:



1. Turn on the "OK to cap" signal
2. Activate all cappers where bulbs are required
3. Wait (total still period - capping time)
4. Turn off the "OK to cap" signal
5. Wait (capping time)
6. Move to next position

```
PROCESS CLASS FILAMENTCHAIN;

BEGIN ...
NEXTCYCLE:
          Turn on the "OK to cap" signal;
          FOR each capper DO
              IF capping is possible THEN activate the capper;
          HOLD (total still period := capping time);
          turn off the "OK to cap" signal;
          HOLD (capping time);
          HOLD (move time);
          details of the move;
          GO TO NEXTCYCLE;
END **** FILAMENTCHAIN **** ;
```

After further refinements, we can write our simulation which has the skeleton

```
SIMULATION
    BEGIN
        PROCESS CLASS CAPPER ..........;
        PROCESS CLASS PICKER ..........;
        PROCESS CLASS FILAMENTCHAIN ....;
        generate components;
        set them in motion;
        HOLD(simulation period);
        report;
    END
```

A complete listing is given in Appendix A.

In execution a SIMULA program may be considered as a set of components, each as general as a fully fledged program, and each a structure taken directly from the class declarations. A snapshot at any time would give a picture where each component is "at" a certain statement. This statement is referenced by a local sequence control which references how far down its action sequence we have executed. Only one component is active at a time, the rest are frozen. The synchronisation between the components is achieved by calls on the scheduling procedures, e.g. ACTIVATE, PASSIVATE.

The program shows how a fairly complex program may be split into separate components, and then each one modelled on its own. The idea of writing components by describing their actions in English first has proved very useful in practice. The NCC SIMULA courses use this idea heavily to teach simulation, and it is possible to now "execute" paper and pencil models, following the instructions in English, and check upon the synchronisation of actions before proceeding to the detailed coding.

Future

For the future, there are three major extensions which would be desirable extensions to current SIMULA Systems.

1. External classes

All current systems allow for external procedures, but not external classes. Implementing the latter involves extra problems, for whereas procedures are self-contained action clusters which may not be referenced from without, objects of classes declared in prefix layers certainly will be referenced and accessed. The internal class names and their attributes must therefore be known to a program when it absorbs an external class. However, the fact that external compilation must come as soon as funds permit was taken note of when the current implementations were written, and their incorporation should not prove too difficult.

2. External objects

This means the rolling out of passive objects under program execution, and rolling them back in when they are either to be active again or accessed. This would allow much larger problems to be run. So far as I know this problem has not been faced squarely by any current implementation. Its implementation will require addressing via a look-up table rather than by address in core, and possibly the serial numbering of all classes during the lifetime of a compiler.

3. DBTG Proposals

The DBTG proposals are being examined by groups at NCC and the University of Oslo to see how SIMULA can be hooked onto data bases. A provisional design proposal has already been worked out and was presented at Nord-DATA 1972 [14].

References

[1]. O-J. Dahl and K. Nygaard:
SIMULA - A Language for Programming and Description of Discrete Event Systems. Introduction and User's Manual.
NCC Publ., August 1969.

[2]. Ole-Johan Dahl, Bjørn Myhrhaug and Kristen Nygaard:
SIMULA 67 Common Base Language
NCC Publ. S22, April 1971.

[3]. SIMULA for IBM-360: Users Guide
NCC Publ. S23, May 1971.

[4]. SIMULA for IBM-360: Programmers Guide
NCC Publ. S24, May 1971.

[5] EXEC 8 UNIVAC SIMULA Users Guide
NCC Publ. S36, August 1971.

[6] EXEC 8 UNIVAC SIMULA Programmers Reference Manual
NCC Publ. S37, May 1972.

[7] K. Babcicky and P. Wynn:
Special features of the IBM 360/370 SIMULA System
NCC Publ. S41, August 1972.

[8] Control Data 6400/6500/6600
Computer Systems SIMULA References Manual
Control Data Corporation, 1969.

[9]. G.M. Birtwistle, O-J. Dahl, B. Myhrhaug and K. Nygaard:
SIMULA BEGIN
Studentlitteratur, Sweden, 1973.

[10] K. Nygaard:
System Description in SIMULA - an Introduction
NCC Publ. S35, November 1970.

[11] Antti Virjo (Finnish State Computer Centre):
A Comparative Study of Some Discrete Event Simulation Languages
(Lecture delivered at NordDATA 72, Helsinki).
NCC Publ. S43, August 1972.

[12] NCC System Programming Group
The Structure of the NCC SIMULA compilers and Bench Mark Comparisons with other Major Languages
(Lecture delivered at NordDATA 72).
NCC Publ. S43, August 1972.

[13] Th. S. Kjeldaas:
CLASS DRAUGHTING - the description of a SIMULA application package for geometrical constructions.
NCC Publ. S39, June 1972.

[14] H. Hegna:
SIMULA for databaser (in Norwegian) NordDATA 72.
Proceedings of the NordDATA conference, Helsinki 1972.

Acknowledgements

The author would like to thank O-J. Dahl and S. Slemmons for very helpful discussions during the preparation of this paper. Further thanks are due to the typing pool and printing shop of NCC for their usual speedy and accurate work.

Ellison : Do you know of any other implementations?

Birtwistle : The University of Oslo and the Norwegian Defence Research Establishment have a Cyber 74 and the latter are re-implementing SIMULA for the big CD range in PASCAL. 2/3rds of the system will thus go on to the ICL 1900's ... . CII in Paris have implemented SIMULA on the CII 10070 and IRIS 80. Mr. Wayne Milner has written an interpreter for Burroughs. Finally the Swedish Defence Research Establishment are to begin an implementation in 1973 for their virtual memory PDP-10.

Pyle : If you make mistakes in a simulation, like leaving out a HOLD statement, what happens?

Birtwistle : You go on without it - presumably winding up with unexpected results. While on the subject of simulation, I would like to point out that SIMULA is not restricted to the process mode of description, but can easily be used for the activity mode too. An activity always has the skeleton:

```
PROCESS CLASS ACTIVITY;
   IF conditions are met THEN
      BEGIN seize entities needed for task;
            HOLD (task time);
            release entities used in task;
      END;
```

When the synchronisations are tricky, the activity model will be the easier to write, but less efficient.

Moore : Have you considered an object as data and looked at DDL's in this way?

Birtwistle : Personally, not seriously. But the group mentioned above in Future 3 are doing just that.

Appendix A

Listing of the program. The problem was given to the author by Dr. R. Dietvorst of Philips, Holland.

```
ENT**********************************************************************
**********************************************************************
                                                                      **
      PROBLEM FROM PHILIPS IN EINDHOVEN : BULB CAPPING                **
                                                                      **
      TYPICAL INPUT:                                                  **
      BASIC TIME UNIT T  :    0.16                                    **
      NUMBER OF MACHINES :       6                                    **
      SIMULATION PERIOD  :    3600                                    **
      A RANDOM SEED FOR EACH MACHINE                                  **
                                                                      **
**********************************************************************
**********************************************************************;

EGER N,     COMMENT***NUMBER OF MACHINES;
     M;
L THREET,FIVET,T;
INREAL;
EET:=3*T;
ET :=5*T;
ININT;
N+1;

ULATION
IN
 BOOLEAN ARRAY FILAMENTWITHOUTBULB(0:M);
 REF(PICKER)ARRAY P(1:N);
 REF(CAPPER)ARRAY C(1:N);
 INTEGER NEWFILAMENTS,UNCAPPED,I;
 REAL SIMPERIOD;
 BOOLEAN CAPPERSCANNOWCAP;

 PROCESS CLASS FILAMENTCHAIN;
 BEGIN
    INTEGER I;

    NEXTCYCLE:
    NEWFILAMENTS:=NEWFILAMENTS+1;
    CAPPERSCANNOWCAP:=TRUE;
    FOR I:=1 STEP 1 UNTIL N DO
    IF FILAMENTWITHOUTBULB(I) THEN ACTIVATE C(I);
    HOLD(THREET);

    CAPPERSCANNOWCAP:=FALSE;
    FOR I := M STEP -1 UNTIL 1 DO
    FILAMENTWITHOUTBULB(I):=FILAMENTWITHOUTBULB(I-1);
    IF FILAMENTWITHOUTBULB(M) THEN UNCAPPED:=UNCAPPED+1;
    HOLD(FIVET);
    GOTO NEXTCYCLE;
 END***FILAMENT CHAIN***;
```

```
CESS CLASS ARM(I); INTEGER I;
IN
BOOLEAN FULL;
REF(ARM)PARTNER;
***ARM***;

 CLASS PICKER;
IN
INTEGER U;

U:=ININT;

NEXTBULB:
FULL:=FALSE;
HOLD(UNIFORM(7,11,U));

FULL:=TRUE;
IF PARTNER.FULL THEN PASSIVATE ELSE
BEGIN
   PARTNER.FULL:=TRUE;
   ACTIVATE PARTNER;
END;
GOTO NEXTBULB;
D***PICKER***;

M CLASS CAPPER;
GIN
 INTEGER CAPPED;

 FULL:=TRUE;

 NEXTTRY:
 WHILE ¬ ( CAPPERSCANNOWCAP AND FULL AND FILAMENTWITHOUTBULB(I) ) DO
   PASSIVATE;
 FILAMENTWITHOUTBULB(I):=FALSE;
 CAPPED:=CAPPED+1;
 HOLD(THREET);
 IF PARTNER.FULL THEN ACTIVATE PARTNER ELSE FULL:=FALSE;
 GOTO NEXTTRY;
ND***CAPPER***;
```

```
SIMPERIOD:=INREAL;

FILAMENTWITHOUTBULB(0):=TRUE;
FOR I:=1 STEP 1 UNTIL N DO
BEGIN
   C(I):-NEW CAPPER(I);
   P(I):-NEW PICKER(I);
   C(I).PARTNER:-P(I);
   P(I).PARTNER:-C(I);
   ACTIVATE C(I);
   ACTIVATE P(I);
END***INITIALISATION***;

ACTIVATE NEW FILAMENTCHAIN;
HOLD(SIMPERIOD);

OUTTEXT("CAPPER  CAPPED");
OUTIMAGE;
FOR I:=1 STEP 1 UNTIL N DO
BEGIN
   OUTINT(I,6);
   OUTINT(C(I).CAPPED,8);
   OUTIMAGE;
END;
OUTINT(UNCAPPED,6);       OUTTEXT(" FILAMENTS GOT THROUGH UNCAPPED");
OUTIMAGE;
OUTINT(NEWFILAMENTS,6);   OUTTEXT(" WAS THE TOTAL OF NEW FILAMENTS");
OUTIMAGE;
END;
```

```
CAPPER  CAPPED
     1     405
     2     399
     3     402
     4     401
     5     398
     6     398
   406 FILAMENTS GOT THROUGH UNCAPPED
  2813 WAS THE TOTAL OF NEW FILAMENTS
```