An experiment in the reduction of references.

C. A. R. Hoare.

Summary. This paper explores how far it is possible to retain the excellent program and data structuring capabilities of SIMULA 67 . in a high-level programmi language which does not contain the problematic concept of a reference. The examples of "Hierarchical Program Structures" [1] are used as experimental material.

Introduction.

I have argued [illegible] [illegible] that a significant [illegible] should not contain the [illegible] concept [illegible] (as the [illegible] [3]). FORTRAN ([illegible] the [illegible] of [illegible] [4]). [illegible] record class [illegible] [5] [illegible] PASCAL [6]. [illegible] is [illegible] a general relational network [illegible]. This application of references will not be discussed further. The case against more general kinds of reference is made more fully in [7].

But there would seem to be other legitimate uses for references; for example, in SIMULA 67 [8], they are deeply rooted in the class concept, which provides a powerful method for structured data representation and for quasiparallel programming (coroutines). The question therefore arises, how far is it possible to reproduce these excellent features of SIMULA 67 in a language which does not contain the concept of a reference? This paper explores an answer to this question by

B suggesting a slight change to SIMULA 67, which effectively removes referencing from it, and showing that nearly all of the examples of [1] can still be successfully programm

~~It appears that most~~

~~of the [illegible] can be [illegible] for~~
~~[illegible] [4] (and~~
~~[illegible] [2],~~

(provided that the concept of a condition [10] is introduced into the language.

Changes to SIMULA 67.

The main change that I would suggest to SIMULA 67 is to the definition of the special reference assignment :-, and the special 'reference' tests, == and =/=. These should be implemented in such a way that they behave logically like the normal assignments := and the normal tests of equality and inequality; ~~and~~ and consequently, the ~~that the normal~~ familiar program proving methods should ~~can~~ be applicable to them. The implementation may still use references as indirect addresses in representing variables of a class; but it must ensure that each object of the class is pointed to by one and only one reference variable. It is the uniqueness of referencing that avoids the logical and practical problems of "sharing", which arise in a language which uses references (in a more essential way.

A typical reference assignment

r := e

will be implemented as follows:

(1) All records (if any) directly or indirectly pointed to by r are returned to the free storage pool.

(2) A copy of all records directly or indirectly pointed to by e is taken.

(3) The address of this fresh copy is planted in r.

In practice, there will usually be some significant shortcut, which gets rid of the copying. For example, nearly all occurrences of := in [4] are in the context:

```
r := new classobject ... ;
```

where r at the time has the value none. In such a case, the implementation can be identical to the current implementation of SIMULA 67.

Copying can also be inhibited when [illegible] both the sides of the assignments are:

```
sub := sub.right;
```

? pp. 200-1 in [4].

Here, the effect is to replace sub by a part of what it was originally pointing to, and it can be implemented (without copying) by simple overwriting of the sub variable, after the record immediately referenced by sub has been returned to the free pool.

The operations == and ≠ are implemented by comparing the contents of all (non-reference) components of the records (directly or indirectly) referenced by both sides. This is the definition of the LISP equals function and has all the usual logical properties of equality.

One consequence of the avoidance of sharing is that it is no longer necessary to use a scan-mark garbage collector for recovering free store. An area of store is known to go out of existence on the deallocation or overwriting of a reference to it. Thus on block exit, all records directly or indirectly referenced from the activation record of the block should ~~can~~ be returned by the block postlude to free store. The explicit return of freed storage on the instant that it goes out of use can be very much more efficient than waiting for it to be collected by a scan-mark garbage collector. However for functional languages like LISP, and for many symbol manipulation problems, the use of shared storage is ~~to~~ be ~~highly recommended~~, preferred, since garbage collection will be much cheaper than repeated copying. However, as explained in [7] this use of references and sharing can also be hidden from the user of a high level language.

Since the proof rules for referenced variables and normal variables are the same, there is no need to use the special notations :-, ==, =/=, or =/=, and they could be replaced by the more familiar notations :=, =, ≠, and perhaps var. Also, the keyword new could perhaps be omitted on a record generator.

However, in the examples of this paper I shall retain the SIMULA notation, on the understanding that it has the new meaning.

The redefinition of the effect of reference assignment does not affect any of the programs in the first ~~up to~~ five chapters of [1]; in fact it probably improves the quality of their implementation.
The first impossible program is the two-way list (TWLIST, page 206), which is utterly dependent on the "sharing" of records between several references.
Although TWLIST is used by MIMSIM (page 212), and hence by LEE and JOBSHOP, we shall attempt to show an alternative way of programming these simulations, using the synchronising concept of a "condition", described in [10].

A condition is a type of quantity, declarable in the normal way

condition $x, y, z;$

However, there is no stored value associated with a condition, and the only operations which can be performed upon conditions are the following:

x.wait — suspend the current class object at the end of the queue of objects also waiting on condition x; on resumption, this object will proceed with the next following command.

x.signal — resume the first class object waiting on condition x, and pass control directly to its next command. The signalling object suspends itself (on an anonymous stack) until the newly resumed object (and all objects directly or indirectly created or signalled by it) are either waiting again, or have finished. If there was no object waiting on x, signal has no effect.

x.wait(p). Same as x.wait, except that waiting objects are queued in order of their parameter p; and on a signal, it is the object with *lowest* p that is resumed. This is known as a scheduled wait.

x.queue   a Boolean function which yields the value true if there is an object waiting on x, and false otherwise.

These definitions are identical to those which are suggested in [10] as appropriate for the synchronisation of genuine parallel processes. However, in a quasiparallel environment, there is no need for mutual exclusion of critical regions, since there is only one process, which is scheduled under the explicit control of the programmer. Thus there is no need for a monitor concept. Proof rules for conditions are given in [10], but it is difficult to see them as relevant to simulation programs, where the history of the computation ~~is not~~ is more important than the final state of the program variables.

As a simple example of the use of conditions, consider the routine group class of the JOB SHOP in [1] page 218. The ref(list) Q is simply replaced by a condition variable.

```
class machine group (nm); integer nm;
  begin condition Q;
    procedure request;
      begin nm:=nm-1; if nm<0 then Q.wait end;
    procedure release;
      begin nm:=nm+1; if nm≤0 then Q.signal end;
  end machine group;
```

The MINISIM class can be programmed in a manner very similar to the alarm clock monitor of [10]. It is simpler than the MINISIM because, as seen in the example above, activate and wait are unnecessary; furthermore, the start parameter of simulate can be omitted, a single global time is adequate for all processes, and the process class is also unnecessary. However, it is useful to postulate a facility "stop" which terminates the current object, as though by a jump to its end. The sequencing set SQS is implemented simply by a scheduled condition.

```
class MINISIM;
begin real time, limit;
      condition SQS;
   procedure hold (T); real T;
   if T > 0 then
      begin real alarm;
            alarm:= time + T;
            if alarm > limit then stop;
            comment stop is intended to terminate
               the current object;
            SQS.wait(alarm);
            time:=alarm;

      end;
   procedure simulate (finish);
             real finish;
      begin limit:=finish;
            while SQS.queue do SQS.signal;
comment the next signal
will not be given until
the activity resulting
from the previous
one has quiesced;
      end simulate;

end MINISIM;
```

A   As an example of the use of these facilities, consider the order class in the jo JOBSHOP of [1]

```
class order (n); integer n;
begin integer array mg[1:n]; array pt[1:n]; integer s;
      hold (inreal - time);
      for s := 1 step 1 until n do
      begin mg[s] := inint; pt[s] := inreal end;
      if ¬ last item then new order (inint);
      comment creates a new order and passes control to it.
      This order resumes when the new order quiesces;
      for s := 1 step 1 until n do
      inspect mgroup[mg[s]] do
            begin request; hold(pt[s]); release end;
      comment the reference variable M has been
         removed by the use of the inspect feature
         of SIMULA 67
end of order;
```

The complete JOBSHOP program is:

```
begin integer nmg; nmg:= inint;
MINISIM begin
class machinegroup ... as shown above...;
ref (machinegroup) array mgroup [1:nmg];
class order ... as shown above...;

 integer k; real lim; lim:= inreal;
 for k:= 1 step 1 until nmg do mgroup[k]:-new machinegroup(inint);
  new order (inint);
  comment control returns here when this new order
      does SQS.wait (inside hold);
  simulate (limit);
  comment this will reactivate the first waiting
      object on SQS whenever there is nothing else to do;

  end MINISIM
end JOBSHOP;
```

This version of the JOBSHOP seems simpler to write, and more straightforward in execution than one based on the TWLIST concept.

Back to coroutines.

The introduction of conditions was made necessary by the impossibility of coding two-way lists without explicit references. Of course this extension of the built-in facilities of a language (especially a language with good 'self-extension' facilities) is more an admission of failure than (a laudable achievement). However, perhaps the failure can be redeemed if it can be shown that some existing feature of the language can now be removed. It turns out that the coroutine features of SIMULA 67 ~~can in fact~~ (detach, call, and probably also resume) can be replaced by the condition concept.

To do this, declare in some suitable place a condition for each coroutine which contains a detach; then replace each detach instruction by a wait on this condition, and replace each call (of the coroutine) by a signal on the condition. As an example consider the permuter on page 191 of [1]; the typical structure for a program (which uses it) might be:

```
ref (permuter) P;
P :- new permuter (N);
while P.more do
    begin ... inspect P.p ... P.cond.signal end;
```

Then the permuter class would have the structure:

```
class permuter(n); integer n;
    begin integer array p[1:n];
          Boolean more; condition cond;
              ... replace "detach" by cond.wait ...

    end
```

A similar technique may be used to replace the "resume" instructions of the text transformation program(s) (page 128. Declare

condition w1, w2;

and ensure that each "producer" of a character in c1 waits on w1 until it has been "consumed", and similarly for w2 and c2. The corresponding signals are given by the "consumer", when he wishes c1 or c2 to be refilled.

```
class  pass 1;
          while true do
          begin integer i;   incard;
               for i:=1 step 1 until 80 do
               begin c1:= C[i]; w1.wait end;
               c1:= blank;  w1.wait
          end infinite loop;
class  pass 2;
~~pass2.~~ while true do
          begin if c1 = "*"
            then begin w1.signal;
                      if c1 = "*" then c2:="↑"
                      else begin c2:= "*"; w2.wait;
                                c2:=c1
                      end
            end
          else c2:= c1;
          w2.wait; w1.signal
          end pass2
```

```
class pass 3;
   while true do
      begin integer i;
         for i := 1step 1 until 125 do
            begin L[i] := c2;
                  if c2 = "end"
               then begin for i := i+1 step1 until 125 do
                                 L[i] := blank;
                          lineout;
                          stop; comment the main program
                                   will now be resumed;
                    end
               else w2.signal
            end; this line;
         lineout.
      end pass 3;
```

The main program can now be written;

```
begin  disassembler :- new pass 1;
       squasher :- new pass2;
       assembler :- new pass3;
end;
```

This is probably not the most elegant program that could be written for its purpose, but it is the one most similar to the program displayed in [1].

Conclusion.

This paper has suggested that many of the excellent program and data structuring features of SIMULA 67 can be reproduced in a language which does not contain the problematic concept of a reference. Several of the examples of [1] have been reprogrammed in such a language, and all the others could be so reprogrammed, with the exception of TWLIST and possibly "find" (in the binary search tree). However, references ~~will probably~~ ~~still be required~~ may still be used in the implementation of such a language. Furthermore, the concept of a record class may still be required to ~~program~~ represent relational networks, like those of cities and roads in the LEE program on page 215 of [1] (although these can often be represented satisfactorily as integers and arrays).

This paper owes much to the kind assistance of J. Bezivin, J. Kaubisch, R.H. Perrott, and O.-J. Dahl.

References.

[1] O.-J. Dahl and C.A.R. Hoare.
Hierarchical Program Structures.
in Structured Programming Academic Press 1972.

[2] C. A. R. Hoare.
Hints on Programming Language Design.
Stanford University CS

[3] A. van Wijngaarden et al.

[4] PL/I

[5] N. Wirth and C. A. R. Hoare
A Contribution to the Development of ALGOL
Commun ACM

[6] N. Wirth.
The Programming Language PASCAL.

[7] C. A. R. Hoare
Recursive Data Structures.
Stanford University CS

[8] Birtwistle et al.
SIMULA BEGIN
Studentlitteratur and Auerbach 1973.

[9]. C. A. R. Hoare.
Proof of Correctness of Data Representations
Acta Informatica.

[10] C. A. R. Hoare
Monitors: an Operating System Structuring Concepts
Stanford University CS

[12] J. McCarthy et al
LISP 1.5.