Summary. This paper explores how it is possible to utilize the excellent program and data structuring capabilities of SIMULA 67, in a high-level programming language, which does not contain the problematic concept of a reference. The examples of "Hierarchical Program Structures" [1] are used as experimental material.
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 in answer to this question by...
suggesting a slight change to ECP [10], which effectively removes referencing from it, and ensuring that nearly all of the examples of [11] can still be successfully program

provided that the correct condition [10] is introduced into the language.
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; that is, that the normal (familiar) program proving methods 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.
In practice, there will usually be some significant overhead, which gets rid of the copying. For example, nearly all versions of LISP restructure lists in the context:

```lisp
r : new consobject --
```

where `r` at this time has the value `null`. In such a case, the implementation can be identical to the current implementation of Section 67.

Copying can also be exhibited when the Lisp machine is in the middle of a Lisp program as

```lisp
setq (consobject) (new object)
```

Here, the effect is to replace one object part of what it was previously pointing to, and it can be implemented (without copying) by simply overwriting of the slot variable, which is the area immediately referenced by and that is then returned to the free pool.

The operations `=` and `#=` are implemented by comparing the contents of all components of the objects 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-work garbage collector for recovering free
store. The area of store is known to go
out of existence as the deactivation or unreferring of a
reference to it. Thus, a block and
all records directly or indirectly referred from
the activation record of the block should be
returned by the block protocol to free stores.
The explicit time of freed storage or the instant
that it goes out of use can be very much
more efficient than waiting for it to be collected
by a scan-work garbage collector. However, for
functional languages like LISP, and for many
symbol manipulation programs, the use of
shoved storage is to be preferred,
since garbage collection will be much cheaper
than repeated copying. However, as explained in
[7] the use of references and sharing can also
be hidden from the user of a high level language.
Since the proof rules for referenced variables
are the same, there is no need to introduce the special notation $\texttt{r} = \texttt{r}, \texttt{w}$,
and they could be replaced by the more
natural notation $\texttt{r} = \texttt{w}$, and perhaps var. Also, the
sharing was found easier to model in the record generator.
Quasi-parallel Programming.

The redefinition of the effect of reference assignment does not affect any of the programs in the first six chapters of [1], in fact it probably improves the quality of their implementation.

The first impossible program is the two-way list (TWO-LIST, page 206), which is utterly dependent on the "sharing" of records between several references. Although TWO-LIST is used by HIMSIM (page 212), and hence by LEE and JOBSHOP, we shall attempt to show an alternative way of programming these simulations, using the synchronization concept of a "condition" described in [10].

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

\[
\text{condition } x, y, z ;
\]

However, there is no stored value associated with a condition; and the only operation which can be performed upon a condition is the following.
x.wait — suspend the current class object at the end of the queue of objects also waiting on condition x. If no releases, the 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 (or an anonymous stack) until the newly resumed object (and all objects directly or indirectly created or signaled 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.
These definitions are identical to those which are suggested in [10] as appropriate for the synchronization of genuine parallel processes. However, in a superparallel 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. Root values for conditions are simpler concepts. Root values for conditions are given in [10], but it is difficult to see them as relevant to computer programs, where the history of the computation is what is more important than the final state of the program variables.

As a simple example of the use of conditions, consider the archive group class of the TOB SHLP in [1] page 213. The ref (list) Q is simply replaced by a condition variable.
class machine group (mm); 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 as seen in the example above, because 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 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 of 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;
    end;

    procedure simulate (finish);
    "
    real finish;
    begin
        limit := finish;
        while SQS queues do SQS.signal;
        repeat the next signal
        will not be given until
        an activity resulting
        over the person 0
        see has patients
        and simulate 1
As an example of the use of these facilities, consider the order class in the JOBSHOP of [1].

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

end of order;
The complete JOBSHOP program is:

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

integer k; real lim; lim := unreal;
for k := 1 step 1 until nmg do mgroup[k] := new machinegroup(limit);
new order (limit);
comment control returns here when this new order
does SQS.wait (inside hold);
simulate (limit);
comment this will reactivate the first waiting
object in SQS whenever there is nothing due to do;
end MINISIM
end JOBSHOP;
```

This version of the JOBSHOP seems simpler
to write, and more straightforward in execution than
me based on the TWIST1 concept.
The introduction of condition 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)—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 by a signal on the condition. As an example consider the parameter on page 191 of [1] the typical structure for a program might be:
\[ \text{move (permu) P;}
\]
\[ \text{P: new permuter (N);} \]
\[ \text{while P.move do}
\]
\[ \text{begin... inspect P.p... P.cond.signal.end;}
\]

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 best transformation (page 128). Declare:

\begin{verbatim}
    condition w1, w2;
\end{verbatim}

and ensure that each "producer" of a character in \( c_1 \) waits on \( w_1 \) until it has been "consumed", and similarly for \( w_2 \) and \( c_2 \). The corresponding signals are given by the "consumer", when she wishes \( c_1 \) or \( c_2 \) to be refilled.
class pass 1:
    while True do
        begin integer i; increment

        for i := 1 step 1 until 80 do
            begin cl := c[i]; w1, wait end;

        end blank; w1, wait
        end infinite loop;

    class pass 2:
        while True do
            begin if cl = "x" then
                begin w1, await;
                if cl = " " then cl := "\n"
                else begin cl := "x"; w2, wait;

                end
            end

            end cl := cl;
            w1, wait; w1, await
        end pass 2
class pass 3;
while true do
  begin integer i;
    begin 1: step 1 until 125 do
      begin L[i] := c3;
        if c2 = "end"
          then begin for i := 1 step 1 until 125 do
                       L[i] := blank;
                      linout;
                      stop; comment: the main program
                      will now be resumed;
                      end
                    end;
    end
  end pass 3;

The main program can now be written:

begin disassembler := new pass 1;
squasher := new pass 2;
assembler := new pass 3;
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 TWIST and possibly "find" (in the binary search tree). However, references will still be used in the implementation of such a language. Furthermore, the concept of a record class may still be required to 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. Kambeide, R.M. Perrott, and O. J. Dool.
References.

    Hierarchical Program Structures.

    Hints on Programming Language Design.
    Stanford University CS


    A Contribution to the Development of ALGOL
    Commun ACM

    The Programming Language PASCAL.

    Recursive Data Structures.
    Stanford University CS
Proof of Correctness of Data Representations
Acta Informatica.

Monitors as Operating System Structuring Concepts
Stanford University CS

LISP 1.5.