A synthesis emerging?

Introduction.

This document does not contain language proposals; at a later stage they may be inspired by it. It has no other purpose than the recording of discussions and programming experiments. It is exciting because it seems to open the possibili of writing programs that could be implemented

a) either by normal sequential techniques
b) or by elephants built from mosquitos
c) or by a data-driven machine.

That programs intended for the second or third implementation could be "inefficient" when regarded as sequential programs, is here irrelevant. The important result would be that the same mathematical technique for the intellectua mastery of sequential programs can be taken over --hopefully lock. stock and barrell-- for the intellectual mastery of those, as yet less familiar designs. Finally --and that seems the most important promise-- it introduces the possibilit of concurrent execution in a non-operational manner.

From the past, terms as "sequential programming" and "parallel programming" are still with us, and we should try to get rid of them, for they are a great source of confusion. They date from the period that it was the purpose of our programs to instruct our machines: now it is the purpose of the machines to execute our programs. Whether the machine does so sequentially, one thing at a time, or with a considerable amount of concurrence, is a matter of implementation and should _not_ be regarded as a property of the programming language. In the years behind us we have carried out this program of non-operational definition of semantics for a simple programming language that admits (trivially) a sequential implementation; our ultimate goal is a programming language that admits (highly?) concurrent implementations equally trivially. The experiments described in this report are a first step towards that goal.

27th and 31st July, 1975.

It all started on Sunday 27th of July 1975, when Tony Hoare explained to me in the garden of Hotel Sepp in Marktoberdorf (Western Germany) upon my request the class-concept of SIMULA (including the so-called _inner_-concept); at least he explained his version of it. I had always stayed away from it as far as possible, in order to avoid contamination with the extremely operational point of view as practiced by Dahl c.s., and, after some time I could not even (understand their mechanistic descriptions anymore: they just made me shudder. Late 1974 Tony sent me a paper, that looked better, but still made me shudder; I read it once, but doubting whether I could endure the exposure, I cousciously refused to study it at that moment. On Saturday 26th I decided that the moment to be courageous had come, and asked Tony to explain to me what he was considering. He was a tolerant master, allowing me to change terminology, notation and a way of looking at it, things I had to do in order to make it all fit within my frame of mind. To begin with I shall record how our discussions struck root in my mind. (Whether a real SIMULA-fan still recognizes the class-concept, is something I just don't know: maybe he gets the impression that I am writing about something totally different. My descriptions in what follows are definitely still more operational and mechanistic than I should like them to be: it is hard to get rid of old habits!)

* * *

Suppose that we consider a natural number, which can be introduced with the initial value zero, can be increased and decreased by 1, provided it remains non-negative. A non-deterministic, never ending program, that may generate <u>any</u> history of a natural number is then

```
nn begin privar x; x vir int := 0;
        do true → x:= x + 1
        ▯ x > 0 → x:= x - 1
        od
   end    .
```

Suppose now, that we would like to write a main program operating on two natural numbers y and z , a main program that "commands" these values to be increased and decreased as it pleases. In that case we can associate with each of the two natural numbers y and z a non-deterministic program of the above type, be it, that the non-determinacy of each of these two program executions has ot be resolved ("settled", if you prefer) in such a way that the two histories are in accordance with the "commands" in the main program. For this purpose we consider the following program. (Please remember that the chosen notations are not a proposal: they have been introduced only to make the discussion possible!)

```
nn gen begin privar x; x vir int := 0;
             do ?inc → x:= x + 1
              ▯ x > 0 cand ?dec → x:= x - 1
             od
       end

main program
       begin privar y, z;  y vir nn; z vir nn;
                     :
             y.inc; ...; y.dec; ...; z.inc; ...; z.dec; .....
       end
```

<u>Notes.</u>

1) We have written two programs, eventually we shall have three sequential processes, two of type "nn" --one for y and one for z -- and one of type "main program". The fact that the first one can be regarded as a kind of "template" I have indicated by writing <u>gen</u> (suggesting "generator") in front of its <u>begin</u>.

2) The main program is the only one to start with; upon the initialization "y <u>vir</u> nn" the second one is started --and remains idling in the repetitive construct-- , upon the initialization "z <u>vir</u> nn", the last one is introduced in an identical fashion. It is assumed --e.g. because the "main program" is written after "nn"-- that the main program is within the lexical scope of the identifier "nn".

3) The two indentifiers inc and dec -preceded in the text of nn by a question mark-- are subordinate to the type nn , that is, if y is declared and initialized as a variable of type nn , the operations inc and dec -- invoked by "y.inc" and "y.dec" respectively-- are defined on it and can be implemented by suitable synchronizing and sequencing the execution of the y-program with that of the main program.

4) When in the main program "y.inc" is commanded, this is regarded in the y-program as the guard "?inc" being true (once). Otherwise guards (or guard components) with the question mark are regarded as undefined. Only a true guard makes the guarded statement eligible for execution.

5) The block exit of the main program, to which the variables y and z are local, implies that all the "query guards" are made false: when ?inc and ?dec are false for the y-program, the repetitive construct terminates and that local block exit is performed: the "x" local to the y-program may cease to exists. It is sound to view the implicit termination of the blocks associated with the variables y and z to be completed before block exit if the block to which they are local --the main program-- is completed. (End of Notes.)

* * *

In the preceding section we have assumed that the main program was somehow within the scope of "nn". But one can argue what funny kind of identifier this is: on the one hand it is the name of a program text, there are, however, as many nn's as the main program introduces natural numbers. The decent was to overcome this is to introduce a fourth program, one "natural number maker" say peano . Suppose that it is the purpose of peano not only to provide --i.e. to create and to destroy-- natural numbers, but also to print at the end of its life the maximum natural number value that has ever existed.

```
peano
begin privar totalmax; totalmax vir int := 0;
   do ?nn → gen begin privar x, localmax;
                 x vir int, localmax vir int := 0, 0;
                (//do ?inc → x:= x + 1;
                            do localmax < x → localmax:= x od
                  ▯ x > 0 cand ?dec → x:= x - 1
                 od//);
                 do totalmax < localmax → totalmax:= localmax od
                end
   od;
   print(totalmax)
end
```

```
main program
begin privar y, z;  y vir peano.nn; z vir peano.nn;
         .
         .
         .
      y.inc; ...; y.dec; ...; z.inc; ...; z.dec
end
```

The idea was, that the program called peano is read in and started, until it gets stuck at the repetitive construct with the (undefined) query "?nn". With the knowledge of the identifier peano (and its subordinate peano.nn) the main program is read in and started, and because inc is subordinate to peano-nn, it becomes subordinate to y by the initializing declaration "y vir peano.nn".

Notes.

1) In the above it has not been indicated when peano will terminate and print the value of totalmax.

2) The generator describing the natural number exists of three parts:
its opening code;
(// its local code//);
its closing code .
Only in the opening code --here the facility is not used and in "nN" the "(//" could have been moved forward-- and the closing code access to totalmax , the local variable of peano is permitted. Different natural numbers may "inc"

simultaneously, only their opening and closing codes are assumed to be performed in mutual exclusion.

3) If the main program is a purely sequential one, y.dec immediately after initialization will cause the main program to get stuck. If the main program consists of a number of concurrent ones, the one held up in y.dec may procedd after another process has performed y.inc. Our natural numbers would then provide an implementation for semaphores!

4) It is now possible to introduce, besides the peano given above, a "peanodash" that, for instance omits the recording of maximum values. The main program could then begin with

<u>begin</u> <u>privar</u> y, z; y <u>vir</u> peano.nn; z <u>vir</u> peanodash.nn; .....

The importance of the explicitly named "maker" in the declaration/initialization lies in the fact that it allows us to provide alternative implementations for variables of the same (abstract) type. (End of Notes.)

The above records the highlights of Sunday's discussion as I remember them. Many of the points raised have been recorded for the sake of completeness: we may pursue them later, but most of them not in this report, as the discussion took another turn on the next Thursday.

* * *

On Thursday a couple of hours were wasted by considering how also in the local code onstances of generated processes --natural numbers-- could be granted mutually exclusive access to the local variables of their maker. Although we came up with a few proposals of reasonable consistency, Tony becamse suddenly disgusted, and I had to agree: the whole efforts had been "to separate" and now we were re-introducing a tool for fine-grained interference! Our major result that day was the coding of a recursive data structure of type"sequence". The coding is given on page EWD508 - 4 (omitting the type of parameters and function procedures). It is not exactly the version coded on that Thursday afternoon, but the differences are minor.

It is a recursive definition of a sequence of different integers. Let s be a variable of type sequence.

s.empty	is a boolean function, true if the sequence s is empty, otherwise false
s.has(i)	is a boolean function with an argument i of type integer; it is true if i occurs in the sequence, otherwise false
s.truncate	is an operator upon s, which also returns a boolean value; if s is nonempty, the last value is removed and the value true is returned, if s is empty, it remains so and the value false is returned
s.back	is an operator upon s that returns a value of type nint (i.e. the integers, extended with the value nil); is s is nonempty, the first value is returned and removed from s, if s is empty, it remains so and the value nil is returned
s.remove(i)	is an operator upon s with an argument i of type integer; if i does not occur in s , s is left unchanged, otherwis the value i is removed from the sequence s without changi

```
sequencemaker
begin
do ?sequence →
   gen begin
    (//do ?empty → result := true
        [] ?has(i) → result := false
        [] ?truncate → result := false
        [] ?back → result := nil
        [] ?remove(i) → skip
        [] ?insert(i) →
            begin privar first, rest;
               first vir nint := i; rest vir sequencemaker.sequence;
               do first ≠ nil cand ?empty → result := false
                [] first ≠ nil cand ?has(i) →
                     if first = i → result := true
                      [] first ≠ i → result := rest.has(i)
                     fi
                [] first ≠ nil cand ?truncate →
                     result := true;
                     begin pricon absorbed;
                        absorbed vir bool := rest.truncate;
                        if absorbed → skip
                         [] non absorbed → first:= nil
                        fi
                     end
                [] first ≠ nil cand ?back →
                     result := first; first:= rest.back
                [] first ≠ nil cand ?remove(i) →
                     if i ≠ first → rest.remove(i)
                      [] i = first → first:= rest.back
                     fi
                [] first ≠ nil cand ?insert(i) →
                     if i ≠ first → rest.insert(i)
                      [] i = first → skip
                     fi
               od
            end
       od//)
       end
od
end
```

- -

the order of the remaining elements in the sequence

s.insert(i) — is a operator upon s with an argument i of type integer; if i does occur in s , s is left unchanged, otherwise s is extended wt the far end with the value i.

(The above is a set of rather crazy specifications: they grew in an alternation of simplification --we started with a binary tree-- in order to reduce the amount of writing we had to do, and complications, when we oecame more ambitious and wanted to show what we could do.)

Note. I am aware of the lousiness of the notation of an operator upon s which returns a value. I apologize for this lack of good taste. (End of Note.)

The sequencemaker is very simple: it can only provide as many sequencies as it is asked to provide; the storage requirements for a sequence are very simple, viz. a stack. (In our rejected example of the binary tree, although lifetimes are, in a fashion, nested, life is not so simple.) The sequencemaker has no local variables (like peano); accordingly, each sequence is simple: its opening and closing codes are empty. The outer repetitive constructs describe the behaviour of the empty sequence: all its actions are simple with the exception of ?insert(i) , as a result of which the sequence becomes nonempty. In an inner block, which describes the behaviour of a sequence that contains at least one element, two local variables are declared: the integer "first" for that one element, and the sequence "rest" for any remaining ones.

It is illustrating to follow the execution of the call "remove(i)". Suppose that i does not occur in the sequence: in that case we have constantly "i $\neq$ first", the task to remove i is constantly delegated to the rest, until it is delegated to an empty rest, fro which --9th line-- remove(i) reduces to a skip. If, however, the value i occurs in the sequence, it occurs in a nonempty sequence and "i = first" is discovered; the command then propagates in the form "first:= rest.back". The last non-empty sequence that performs "first:= rest.back" gets --see 8th line-- the value "nil" from its successor and establishes for itself "first = nil". As a result the repetitive construct in its inner block is terminated, an inner block exit is performed, prior to the completion of which all query-guards for its successor are set false, and its successor performs an exit from its outer block and ceases to exist.

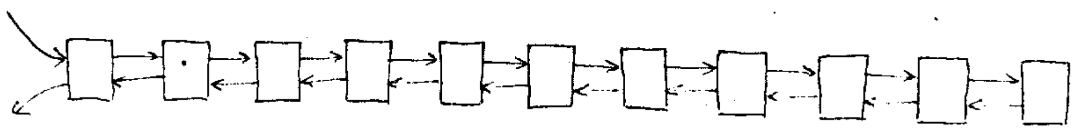
It is also instructive to follow how upon the block exit from

begin privar s; s vir sequencemaker.sequence; .... end

at a moment that s may contain many elements, the sequence s disappears. All query-guards to s are set false, which forces termination of the inner repetitive construct for s which results in block exit from its inner block (which first requires deletion of its rest); upon completion of this block exit, the query-guards still being false, termination of the outer repetitive construct and block exit from the outer block of s are forced. This is very beautiful: the hint to delete itself, given to the head of the sequence, propagates up to its end, reflects there, travels back, folding up the sequence in a nice stack-wise fashion, as, of course, it should. In its elegance --or should I say: completeness?-- it had a great appeal to us.

* * *

It was at this stage, that I realized, that the same program could be visualized as a long sequence --long enough, to be precise-- of mosquitoes:



where each mosquito is essentially a copy of the text between (// and //), and each mosquito is the "rest" for his left-hand neighbour. The execution of the declaration "rest vir sequencemaker.sequence" can be interpreted as a command to one's right-hand neighbour to initialize its instruction counter to the begin of the program. Each mosquito is ready to accept a next command from the left as soon as it has nothing more to do, i.e. its control has successfully returned to one of the sets of query-guards. Giving a command to the right lasts until the command has been accepted when no answer is required, and until the answer has been returned when an answer is required.

It is instructive to follow the propagation of activity for the various commands.

?empty is immediately reflected.

?has(i) propagates up the sequence until i has been detected or the sequence is exhausted, and from there the boolean value (true or false respectively) is reflected and travels to the left until it leaves the sequence at the front end. All the time the sequence is busy and cannot accept another command. The time it takes to return the answer true depends on the distance of i from the begin of the sequence, the time it takes to return the answer false is the longest one, and depends on the actual length of the chain (not on the number of mosquitoes available).

?truncate and ?back propagate in practically full speed to the right: at each mosquito, there is a reflection over one place back to absorb the answer. Note that ?truncate (in the inner block) <u>starts</u> with "<u>result</u>:= true" and ?back <u>starts</u> with "result:= first" --actions which can be taken to be completed when the mosquito to the left has absorbed the value. This is done in order to allow the mosquito to the left to continue as quickly as possible.

?remove(i) propagates still simpler (until it becomes a ?back).

?insert(i) propagates also quite simple, until the wave is either absorbed --because i = first is encountered-- or the sequence is extended with one element. The fascinating observation is that any sequence of ?remove(i), ?insert(i ?back and ?truncate may enter the sequence at the left: they will propagate with roughly the same speed along the sequence, if the sequence is long, a great number of such commands may travel along the sequence to the right. It is guaranteed impossible that one command "overtakes" the other, and we have introduced the possibility of concurrency in implementation in an absolutely safe manner.

<u>Note</u>. Originally truncate was coded differently. It did not return a boolean value, and was in the outer guarded command set

```
?truncate → skip
```

and in the inner guarded command set

```
first ≠ nil cand ?truncate →
   if rest.empty → first:= nil
   ▯ non rest.empty → rest.truncate
   fi
```

As soon as we started to consider the implementation by a sequence of mosquitoes, however, we quickly changed to the code of EWD508 - 4, because the earlier version had awkward propagation properties: two steps forward, one step backward. The version of page EWD508 was coded when we had not yet introduced the type nint; after we had done so, we could also have coded truncate with a parameter of type integer: in the outher guarded command set

```
?truncate(i) → result:= nil
```

and in the inner guarded command set

```
first ≠ nil cand ?truncate(i)→
   result := i; first:= rest.truncate(first)   .
```

The last part of this note is rather irrelevant. (End of Note.)

This was the stage in which we were, when we left Marktoberdorf. As I wrote in my tripreport EWD506 "A surprising discovery, the depth of which is --as far as I am concerned-- still unfathomed."

* * *

What does one do with "discoveries of unfathomed depth"? Well, I decided to let it sink in and not to think about it for a while --the fact that we had a genuine heatwave when I returned from Marktoberdorf helped to take that decision!-- . The discussion was only taken up again last Tuesday afternoon in the company of Martin Rem and the graduate student Poirters, when we tried to follow the remark, made in my tripreport, that it would be nice to do away with von Neumann's instruction counter. (This morning I found a similar suggestion in "Recursive Machines and Computing Technology" by V.M.Glushkov, M.B.Ignatyev, V.A.Myasnikov and V.A.Torgashev, IFIP 1974; this morning I received a copy of that article from Philip H.Enslow, who had drawn my attention to it.)

We had, of course, observed that the propagation properties of "has(i)" are very awkward. It can keep a whole sequence of mosquitoes occupied, all of them waiting for the boolean value to be returned. As long as this boolean value has not been returned to the left-most mosquito, no new command can be accepted by the first mosquito, and that is sad. The string of mosquitoes as shown above, is very much different from the elephant structure that we have already encountered very often, viz. all mosquitoes in a ring.

Nice propagation properties would be displayed by a string of mosquitoes that send the result as soon as found to the right, instead of back to the left! Before we pursue that idea, however, I must describe how I implemented (recursive) function procedures in 1960 --a way, which, I believe, is still the standard one-- .

Upon call of a function procedure the stack was extended with an "empty element", an as yet undefined anonymous intermediate result. On top of that that procedure's local variables would be allocated and during the activation of the procedure body, that location --named "<u>result</u>"-- would be treated as one of the local variables of the procedure. A call

```
?has(i) → if i = first → result:= true
          ▯ i ≠ first → result:= rest.has(i)
          fi
```

could result in 9 times the second alternative and once the first, so that the answer is found at a moment of dynamic depth of nesting, equal to 10. In the implementation technique described, the boolean result is then handed down the stack in ten successive steps: the onymous result at level n+1 becomes at procedure return the anonymous result at level n , that is assigned to the onymous result of level n , etc.: a sequence of alternating assignments and procedure returns. Under the assumption that assignment is not an expensive operation, this is an implementation technique that can very well be defended.

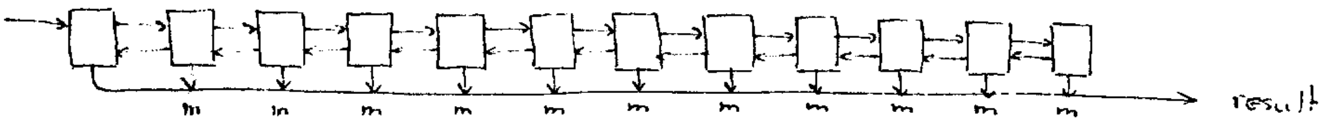
<u>But it is an implementation choice</u>! When implementing

<u>result</u> := rest.has(i)

no one forces us to manipulate the value of "res.has(i)" as an imtermediate result, that subsequently can be assigned! An alternative interface with the function procedure would have been to give it an additional implicit parameter, viz. the destination of the result --e.g. in a sufficiently global terminology, such as distance from stack bottom, say--. In that case the implementation of

<u>result</u> := rest.has(i)

would consist of a recursive call on "has" in which the implicit destination parameter received would just be handed over to the next activation. When, at dynamic depth 10, the boolean value would become known it would instantaneously be placed at its final destination, after which the stack could collapse. Because in the case of a fixed number of mosquitoes, always present, needed or not --that is the simplification I am thinking about now-- there is not much stack collapse, the configuration that now suggests itself is the following



The mosquitoes still have the same mutual interconnection pattern, but I assume that each request for a value, entering the network at the left at the question mark, is accompanied by "a destination" for the result. The reason that I have added the line at the bottom is the following. A sequence is a very simple arrangement, and in that case, also the "external result" as soon as known, could be handed to the right-hand neighbour for further transmission. If, however, we consider the tree that would correspond to a variable of the type "binary tree" , the result would then finally arrive in one of the many leaves. If we associate a real copper wire with each connection between two mosquitoes, and we wish the result to appear at a single point, then we have to introduce some connecting network, such that the various paths of the results can merge. Hence the additional line; the points, marked "m" are binary merge points, we have arranged them linearly, we could have arranged them logarithmicall logically --and perhaps even physically-- we can think of "multi-entry merges".

I am now not designing in any detail the appropriate mechanism for collecting the external result as soon as it has been formed somewhere in the network. My point is that there are many techniques possible, which all can be viewed as different implementation techniques of the same (recursive) program. Their only difference is in "propagation characteristics". The reason that I draw attention to the difference in implementation technique for the sequential machine (without and with implicit destination parameter) is the following. In the case of the linear arrangement of mosquitoes, each mosquito only being able to send to his right-hand neighbour when his right-hand neighbour is ready to accept, we have a pipeline that, by the nature of its construction, produces results in the order in which they have been requested. This, in general, seems to severe a restriction, and for that purpose, each request is accompanied by a "destination", which as a kind of tag accompanies the corresponding result when finally produced. Obviously, the environment driving the network, must be such, that never to requests with the same destination could reside simultaneously in the network.

* * *

True to our principle that about everything sensible that can be said about computing can be illustrated with Euclid's Algorithm, we looked at good old Euclid's Algorithm with our new eyes. We also took a fairly recent version, that computes the greatest common divisor of three positive numbers. It is

x, y, z := X, Y, Z;
<u>do</u> $x > y \rightarrow x := x - y$
▯ $y > z \rightarrow y := y - z$
▯ $z > x \rightarrow z := z - x$
<u>od</u>

with the obvious invariant relation: gcd(x, y, z) = gcd(X, Y, Z) <u>and</u> $x > 0$ <u>and</u> $y > 0$ <u>and</u> $z > 0$.

Our next version was semantically equivalent, but written down a little bit differently, in an effort to represent that in each repetition, it was really the triple x, y, z we were operating upon. That is, we regarded the above program as an abbreviation of

```
x, y, x := X, Y, Z;
do x > y → x, y, z := x - y, y, z
 ▯ y > z → x, y, z := x, y - z, z
 ▯ z > x → x, y, z := x, y, z - x
od .
```

We then looked at it and said: Why only change one value? This, indeed is not necessary, and we arrived at the following --similar. but mathematically different- program:

```
x, y, z:= X, Y, Z;                                  (program 3)
do non x = y = z →
    x, y, z := f(x, y), f(y, z), f(z, x)
od
```

```
with       f(u, v):  if u > v → result := u - v
                      ▯ u ≤ v → result := u
                     fi
```

or, if we want to go one step further for the sake of argument

```
           f(u, v):  if u > v → result := dif(u, v)
                      ▯ u ≤ v → result := u
                     fi
and        dif(u, v): result := u - v   .
```

How do we implement this? We can look at program 3 with our traditional sequential eyes, which means that at each repetition, the function f is invoked three times, each next invocation only taking place, when the former one has returned its answer. We can also think of three different f-networks, which can be activated simultaneously. We can also think of a single f-network, that is activated three times in succesion, but where the comparison of the next pair of arguments can coincide in time with forming the difference of the preceding pair. To be quite honest, we should rewrite program 3 in the form

```
x, y, z:= X, Y, Z;                                  (program 4)
do non x = y = z →
    tx, ty, tz := f(x, y), f(y, z), f(z, x);
    x, y, z := tx, ty, tz
od
```

The reason is simple: we want to make quite clear that always the old values of x, y, z are sent as arguments to the f-network, and we want to code our cycle without making any assumptions about the information capacity of the f-network. The above program works also if we have an f-network without pipelining capacity.

* * *

I was considering a mosquito that would have six local variables, x, y, z, tx, ty and tz; it would first "open" tx, ty and tz, i.e. make them ready to receive the properly tagged results, and then send the argument pairs in the order that pleases it to either one or three f-networksm and would then, as a merge node, wait until all three values had been received. When I showed this to C.S.Scholten, he pointed out to me, that the same result could be obtained by two, more sequential mosquitoes: one only storing the x, y, z values, and another one, storing the tx, ty and tz values, waiting for the three values to be delivered by the f-network. This is right.

Some remarks, however, are in order. I can now see networks of mosquitoes, implementing algorithms that I can also interpret sequentially and for which, therefore, all the known mathematical techniques should be applicable. Each mosquito represents a non-deterministic program, that will be activated by its "query-guards" when it is ready to be so addressed and when it is so addressed, and where the act of addressing in the addressing mosquito is only completed, by the time that the mosquito addressed has honoured the request. We should realize, however, that these synchornization rules are more for safety, than for "scheduling", because dynamically, such networks may have awkward macroscopic properties when overloaded. Take the example of the long string of mosquitoes, that, together form a bounded buffer, each of them cyclically waiting for a value from the left, and then trying to transmit this value to the right. If this is to be a transmission line, it has the maximum throughput when, with n mosquitoes, it contains n/2 values. Its capacity, however is n . If we allow its contents to grow --because new values are pumped in at the left, as long as possible, while no values are taken out at the right, it gets stuck: taking out values from the sequence filled to the brim empties the buffer, but this effect only propagates slowly to the left, and the danger of awkward macroscopic oscillations seems all but excluded.

The next remark is that I have now considered elephants built from mosquitoes, but the design becomes very similar to that of a program for a data-driven machine. The programs I have seen for data driven machines were always pictorial ones --and I don't like pictures with arrows, because they tend to become very confusing--, and their semantics was always given in an operational fashion. Both characteristics point to the initial stage of unevoidable immaturity I now see a handle for separating the semantics from the (multi-dimensional, I am tempted to add) computational histories envisaged. In a sense we don't need to envisage them anymore, and the whole question of parallellism and concurrency has been pushed a little bit more into the domain, where it belongs: implementation. This is exciting.

* * *

A sobering remark is not misplaced either, and that is that we have already considered highly concurrent engines --e.g. the hyperfast Fourier transform via the perfect shuffle-- that seem to fall as yet outside the scope of constructs considered here. And so does apparently the on-the-fly garbage collection. We can only conclude that there remains enough work to be done!

PS. For other reasons forced to go to town, I combine that trip with a visit to the Eindhoven Xerox branch. The time to reread my manuscript for typing errors is lacking and I apologize for their higher density.

25th August 1975
Plataanstraat 5
NL-4565 NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow