Concurrent programming: a preliminary investigation.

(Instead of the usual term "parallelism" I prefer the term "concurrency": parallelism is a term that I would like to reserve for a number of rather identical components, progressing "in parallel", i.e. in rather strict synchronism. The term "concurrency" only refers to the (possibility of) simultaneous activity.)

I observe that rather operational approaches to the problem of defining the semantics of programming languages encounter problems when it is tried to extend them to "concurrent programming languages". The way in which the problem manifests itself is that a set of sequential processes --all by themselves well-understood-- are allowed to operate in a common memory (at some grain of interleaving) and that then the non-determinacy is investigated that is generated by this uncontrolled interleaving. This form of non-determinacy is --not surprisingly-- rather hard to cope with.

What I have done in the past year suggests a rather different attack. In that year I have introduced --not necessarily deterministic-- programs by regarding them as (codes for) predicate transformers. During that stage it is totally irrelevant that the program can also be interpreted as executable code for a sequential machine (that is only a pleasant surprise when we try to implement the programming language the next day). Because initial and final states are by definition connected by a predicate transformer (and only tomorrow, when we consider implementations, by a long chain of intermediate states) nothing in my language is "sequential" as long as I do not drag implementations into the picture. The result is that I do not truly understand anymore what is meant by the usual phrase "parallel programming languages", and that I see only one feasible way of attacking the problem: designing a programming language in my usual way, and then observing (tomorrow) that an implementation with a lot of concurrency is possible. (The language design may silently be motivated by that possibility, but that should not confuse us today, when neither implementation, and a fortiori "time" nor "concurrency" play any role at all.) It is fairly obvious that our programming language should be non-deterministic, but that need not frighten me anymore: it is, after all, now more than a year ago that I decided to regard non-determinacy as the rule and determinacy as a --not very interesting-- special case. This report --of which I hope that it will be followed by others on the same topic-- is only a preliminary survey. (My intention to introduce the possibility of concurrency dates --according to my notes-- back to January 1973, but non-determinacy had to be catered for first --fall 1973--. A new incentive to take up this investigation I received at the IBM-Seminar at Newcastle in September 1974, but I wanted to finish my book first.)

The reversal of an algorithm.

This section records an observation that we made in the fall of 1974 --while we were playing with little programs-- an observation to which we did not pay much attention at the time (it was not even recorded). The other week, when making programs fit for concurrent execution, we were suddenly reminded of it.

Our example dealt with a recognizer for the following syntax:

< sentence > ::= < expression > ;
< expression > ::= < primary > {< operator > < primary >}
< primary > ::= < digit > {< digit >} | (< expression >)

where the braces are to be read as "followed by zero or more instances of the enclosed".

We assume that our recognizer finds the characters of the text in an array variable "in", where "in.low" is the currently "lowest" element --i.e. the one with the lowest index value-- and the operation "in:lorem" removes from "in" this lowest element. (We have separated the reading of the next symbol and the moving of the tape.) The call "sent" will do the recognition --i.e. if "in" does not begin with a sentence, the program will abort-- if we assume the self-explanatory boolean procedures "issemi", "isopen", "isclose", "isop" and "isdigit" with a symbol as argument available.

```
proc sent:  exp; if issemi(in.low) → in:lorem fi corp;
proc exp:   prim; do isop(in.low) → in:lorem; prim od corp;
proc prim:  if isdigit(in.low) → in:lorem; do isdigit(in.low) → in:lorem od
             ▯ isopen(in.low) → in:lorem; exp;
                                           if isclose(in.low) → in.lorem fi

            fi corp      .
```

Here, sent , exp and prim try to remove from the low end of "in" the largest < sentence >, < expression > and < primary > respectively that they can find. If they cannot do so (because in.low is inacceptable) they abort. Note, that there are only three reasons for abortion: a missing semicolon where it is needed (e.g. on the text "123(....") in sent, a missing digit or open parenthesis (e.d. on the text "123+)....") in prim, or a missing closing parenthesis were it is needed (e.g. "(123(...") in prim as well.

The observation we made that the generator of an arbitrary < sentence > --we take the liberty of not bothering about termination-- can be derived from the recognizer in a straightforward manner. (The operations "out:hiext" add symbols at the high end of the array variable "out", which is initially assumed to be empty.)

```
proc sent:    exp; out:hiext(semi) corp;
proc exp:     prim; do ?? → out:hiext(op); prim od corp;
proc prim:    if true → out:hiext(digit); do ?? → out:hiext(digit) od
               [] true → out:hiext(open); exp;
                        out:hiext(close)
              fi corp      .
```

(Here do ?? → S od means "zero or more times S", i.e. it is short for something like

```
goon := true;
do goon → S [] goon → goon:= false od  .
```

This notation with questionmarks has been introduced here for convenience, and also with the intention of presenting recognizer and generator with exactly the same layout.) The transformation from generator to recognizer is equally obvious.

We liked the transformations and said to each other "That is nice, isn't it?", in my short enthousiasm I showed it twice to a colleague, and then we turned to other matters. At that moment we had clearly other interests, for none of us remarked that the output of such a generator could be fed directly as input into a concurrent recognizer.

* * *

Buffering mosquitos.

A configuration that I would like to study at some state of the game is a so-called "elephant built from mosquitos". Mosquitos are little machines and they have a few connections --"legs"-- and an input-leg of one mosquito is paired to the output-leg of another mostquito. When one mosquito transfers information to another one, I shall use in the description of both mosquitos the same name for the connection: if a sending mosquito transmits via leg A the current value of its local variable x , we shall denote this in its text by "A:= x" ; in the description of the receiving mosquito that assigns the transmitted value to its local variable y , reception will be indicated by "y:= A" and for the time being we shall assume that some magic sees to it that these two statements are executed simultaneously, i.e. legs are not supposed to contain memory elements.

A one-place buffer is easily coded as a mosquito: let it have one input leg, A say, and one output leg, B say; then its text can be:

```
do true → x:= A; B:= x od      .
```

This is a fully deterministic mosquito: after the one-place buffer has been filled, we can only empty it and vice versa.

More interesting is a mosquito buffering (FIFO) a maximum of, say, 20 values. Suppose that it has a classical array with twenty elements from buf(0) through buf(19) . In its simplest form --and too naive-- we can code it (with "f" for "filling" and "e" for "emptying")

```
f, e := 0, 0;
do true → buf:(f)= A; f:= (f + 1) mod 20;
 [] true → B:= buf(e); e:= (e + 1) mod 20
od  .
```

But this is very naive, even if we assume that the selection of alternatives is dependent on whether the environment is ready to offer via A and/or to accept via B a next value. We are making rather strong assumptions about the decency of that environment and it seems nicer to make the mosquito itself control its capacity and contents, i.e.

let nf be the number of values accepted via A
let ne be the number of values delivered via B

then the values accepted but not yet delivered should be stored (in order of age) as the values $buf(e)$ through $buf((e + nf - ne - 1) \underline{mod}\ 20)$ and the mosquito itself should maintain

P: $0 \leq nf - ne \leq 20$ which leads to the program

```
f, e := 0, 0; nf, ne := 0, 0;
do nf - ne < 20 → buf:(f)= A; f:= (f + 1) mod 20; nf:= nf + 1
 ▯ nf - ne > 0 → B:= buf(e); e:= (e + 1) mod 20; ne:= ne + 1
od  .
```

We leave to the reader the exercise to convince himself that the following program will do the job as well, provided $K \geq 21$:

```
f, e := 0, 0; n1, n2 := 0, 0;
do (n1 - n2)mod K < 20 → buf:(f)= A; f:= (f + 1)mod 20; n1:= (n1 + 1)mod K
 ▯ (n1 - n2)mod K >  0 → B:= buf(e); e:= (e + 1)mod 20; n2:= (n2 + 1)mod K
od   .
```

(The easiest way to do this seems to maintain first the statements operating on nf or ne and then to prove the invariance of P.)

I would like to make the following remarks about the above programs.
Remark 1. I have not bothered about "termination" but that is so easily fixed that I feel entitled to allow myself the convenience.(End of remark 1.)
Remark 2. From a time-less point of view, the FIFO buffer is strictly deterministic: the output stream equals the input stream. It is only when we consider the sequence of consumptions via A and production via B , that the non-determinism shows itself. When we consider this program as working in a fully cooperative environment --i.e. willing to produce via A and to accept via B when our mosquito feels like it-- the non-determinism of the sequential program displays the complete freedom we envisage (with the possibly meaningless exception that as long as we regard the above executed by a strictly sequential machine, production and consumption will never take place simultaneously.) (End of remark 2.)
Remark 3. Very little prohibits "concurrent execution" of both alternatives. We must make the assumption that "buf:(f)= A" and "B:= buf(e)" will not interfere with each other when tried concurrently, but with $e \neq f$. We must further make the assumption that programs dealing with different variables can be executed concurrently. The only possible form of interference is via

the n's, but with respect to these, our programs have a very special property. If --sequentially interpreted-- the one alternative is selected while the other guard is true, the other guard remains true. And that, of course, implies that we can implement the whole game as two cyclic processes which may be detained during their stream-operations when the environment is not ready and must be detained when the guard is found to be false. (End of remark 3.)

Remark 4. These programs have an even stronger property, but it seems too special perhaps, to make much use of it here: the execution of one alternative is guaranteed to make the other guard true. This means that the one process evaluating (n1- n2)mod K need not even read either the old or the new value of an n with which the other process fools: the evaluation of the guard may be supplied with a rubbish value, a mixture: the only result can be that erroneously the value "false" is evaluated, but then we can say "better luck next time". In this special case we don't even need the mutual exclusion usually provided by a switch. The remark is of significance if we wish to abolish daemons that cater for mutual exclusion in unsynchronized self-stabilizing systems, I think we can drop it here.(End of remark 4.)

Experiments with a sorting elephant.

We consider a long string of mosquitos with between each pair of neighbours a connection L for traffic to the left and a connection R for traffic to the right. The terminology is at first sight perhaps confusing, because now each mosquito has two legs, both called L, but it will turn out to be convenient: in mosquito nr. i

L:=....	means sending to nr. i-1
....:= L	means accepting from nr. i+1
R:=....	means sending to nr. i+1
....:= R	means accepting from nr. i-1 .

Our simplest elephant consists of as many (plus 1 or 2) mosquitos as numbers have to be sorted. We distinguish even and odd mosquitos alternating in the string and do not bother about the fact that the terminal mosquitos have to be slightly different, nor do we bother for the time being about termination.

Initially the even mosquitos contain two numbers, while the odd mosquitos start empty. The even mosquito sorts its contents, sends the smalles value to the left, the largest to the right and then waits (empty) until it has received from both its neighbours a number back and repetatur. The odd mosquito waits until it has received a number from both sides, sorts them and send the smallest to the left and the largest to the right and repetatur.

Let each even mosquito have two variables x and y (for the communication with its lefthand and righthand neighbour respectively). Its repertoire of actions consists of

A:	do $x > y \rightarrow x, y := y, x$ od	(sorting)
B:	$L := x$	(sending to the left)
C:	$x := R$	(receiving from the left)
D:	$R := y$	(sending to the right)
E:	$y := L$	(receiving from the right)

The logically necessary precedence relations can be pictured in the following graph

$$\begin{tikzcd}
 & & B \arrow[r] & C \arrow[dr] & & \\
\text{entry} \arrow[r] & A \arrow[ur] \arrow[dr] & & & A & \text{etc.} \\
 & & D \arrow[r] & E \arrow[ur] & &
\end{tikzcd}$$

and the question is, how to represent them. One way of doing this is with a boolean for each arrow:

```
ca:= true; ea:= true; ab:= false; bc:= false; ad:= false; de:= false;
do ca and ea → A; ca:= false; ea:= false; ab:= true; ad:= true
 ▯ ab        → B; ab:= false; bc:= true
 ▯ bc        → C; bc:= false; ca:= true
 ▯ ad        → D; ad:= false; de:= true
 ▯ de        → E; de:= false; ea:= true
od          .
```

But, although perfectly general, this notation does not attract me very much: it is worse than jumps. Another experiment has been (with "dop" for "do permanently")

```
dop A; par [B]; [C]
         ▯ [D]; [E] rap pod
```

where the squate brackets denote the unit of interleaving, and the <u>par</u> - <u>rap</u> are a sort of <u>parbegin</u> and <u>parend</u>. But this turned out to be a dead alley. (As we shall see shortly, it is awkward for the description of the odd mosquitos.)

I am now mostly attracted --perhaps because it is so new, that I have not seen its shortcomings yet-- by the following notation

```
dop !Ā[A]; [B]; [C]
  ▯ ![A]; [D]; [E]
pod
```

where we have two cyclic processes, and the exclamation mark indicates "mutual coincidence", i.e. the two loops must "share" the single point event A each time. The odd mosquitos can be described similarly: each odd mosquito has two variables u and v , its repertoir consists of

```
F:    do u > v → u, v := v, u od
G:    L:= u
H:    u:= R
J:    R:= v
K:    v:= L
```

and its program is

```
dop [H]; ![F]; [G]
  ▯ [K]; ![F]; [J]
pod      .
```

The charm of the above notation is that we can now combine an even mosquito with its righthand neighbour into a single one, by eliminating the internal traffic, by identifying H(u:= R) and D(R:= y) with

```
IR:   u:= y       ("IR" for Internal to the Right)
```

and by identifying E(y:= L) and G(L:= u) with

```
IL:   y:= u
```

and the formal combination becomes the following program

```
dop ![A]; [B]; [C]
  ▯ ![A]; ![IR]; ![IL]
  ▯ ![IR]; ![F]; ![IL]
  ▯ [K]; ![F]; [J]
pod
```

The next observation is that without any change in its semantics, the middle two lines can be combined:

```
dop ![A]; [B]; [C]
  [] ![A]; [IR]; ![F]; [IL]
  [] [K]; ![F]; [J]
pod
```

The last observation is that, now the alternation of A and F has been nicely expressed by the middle line, that u and y can be the same variable, and that IR and IL disappear, i.e. with

```
A:    do x > y → x, y := y, x od
F:    do y > v → y, v := v, y od
```

the combined mosquito becomes, when all is filled in

```
dop ![A]; [L:= x]; [x:= R]
  [] ![A]; ![F]
  [] [v:= L]; ![F]; [R:= v]
pod
```

The above nicely describes the endless graph with the precedence relations

$$\begin{tikzcd}
A \arrow[r] \arrow[dr] & B \arrow[r] & C \arrow[r] & A \arrow[r] \arrow[dr] & B \arrow[r] & C \arrow[r] & {} & \text{etc} \\
K \arrow[r] & F \arrow[r] \arrow[urr] & J \arrow[r] & K \arrow[r] & F \arrow[r] \arrow[urr] & J \arrow[r] & {} &
\end{tikzcd}$$

Merging two successive mosquitos of the above type leads to

```
dop ![A]; [L:= x]; [x:= R]
  [] ![A]; ![F]
  [] ![A']; ![F]
  [] ![A']; ![F']
  [] [v':= L]; ![F']; [R:= v']
pod
```

Here the primed items refer to the righthand side component. Drawing the beginning of the endless graph with the precedence relations, as I did for the previous mosquito, is an exercise that I leave to my readers. It is quite instructive and gives some idea of the power of this notation. (That very power may also be its greatest weakness, but that is another story.)

Again a buffering mosquito.

The following mosquito describes a buffer with a maximum capacity of three values in our new notation:

```
dop ![a:= L]; ![b:= L]; ![c:= L]
  ▯ ![a:= L]; ![L:= a]
  ▯ ![b:= L]; ![L:= b]
  ▯ ![c:= L]; ![L:= c]
  ▯ ![L:= a]; ![L:= b]; ![L:= c]
pod     .
```

Here I am, playing with a new toy! The top line expresses that the local variables are filled cyclically in the order (a, b, c), the bottom line expresses that they are emptied in the same cyclic order. The second line expresses that with regard to the local variable a filling and emptying alternate (so that everything going in goes out again), the next lines express the same relation for b and c .

Remark. In the above description I have allowed myself a sloppy thing: after the exclamation marks I have written down the statements themselves, instead of their names. We could require in several lines the coincidence of ![S1] and the coincidence of ![S2] , while elsewhere it is defined that both S1 and S2 are, for instance, "skip". (End of remark.)

Although I think the above mosquito intriguing, I should not close my eyes to the fact that I currently don't see, how to derive from the above a buffering mosquito in the style of EWD476 - 4. If this inability is not overcome in an acceptable manner, that fact may point to an, as yet unfathomed, weakness of the notation. I hope that time will show, either in one way, or in the other.

Multiple coincidence.

It seems unnatural to restrict coincidence to two occurrences, more precisely, to the occurences in just two cyclic processes. In the elephant for the hyper-fast Fourier transform, with the so-called "perfect shuffle", each mosquito starts broadcasting its contents z via the legs 01 and 02 to two different mosquitos, simultaneously waiting to receive via I1 and I2 to values from two different mosquitos, accepting these values in the local variables x and y. After all this information exchange, it performs basically

F: z:=f(x, y). This would lead to the structure

```
dop [O1:= z]; ![F]
  ▯ [O2:= z]; ![F]
  ▯ [x:= I1]; ![F]
  ▯ [y:= I2]; ![F]
pod          .
```

It seems <u>unattractive</u> to reduce the number of four cyclic processes to two by imposing more stringent sequencing constraints upon the contacts with the mosquito's outer world. First of all we must be careful not to introduce global deadlocks, such as would be caused by

```
dop [O1:= z]; [x:= I1]; ![F]
  ▯ [O2:= z]; [y:= I2]; ![F]
pod
```

as this would start all mosquitos sending and none in the mood to receive. Secondly, we would like all mosquitos to be of identical structure, and for two of the mosquitos one of their own output legs is connected to one of their own input legs. Obviously I do not care too much about the analysis which sequencing constraints to be imposed upon the external contacts are still admissible. Thirdly, it would reduce the traffic density between the mosquitos, so time-wise it is not attractive either. Perhaps it would be wise to indicate at the exclamation marks the multiplicity of coincidence. The synchronization structure of the mosquitos of the hyper-fast Fourier transformer is mentioned here as a recording of the issues involved in elephant construction.

<u>Things to be done sooner or later</u>.

1) Instead of falling in love with my new notation, I should continue the investigation of alternatives.

2) The current notation has to be extended so as to include a termination criterion as well.

3) It should be investigated whether the current notation should be ex-

tended so as to include other reasons for delay besides (internal or external) coincidences. I would feel safer about it, if I could find an argument that this extension is not necessary, for that would do away with the analysis of obligations for waking up.(Is it, therefore, too much to hope for?)

4) I have not given yet a formal definition of the semantics of a single mosquito. Only thereafter can I hope to derive theorems about merging several mosquitos into a single one.

5) It should be tried on some standard exercises, such as The Dining Philosophers, and perhaps also on the Readers and the Writers.

6) It should be discovered, whether proving something about an elephant is more conveniently done by merging its mosquitos into a single giant mosquito --which is equivalent to the elephant-- or, whether we prove our assertions about the elephant in two steps: first proving everything about the individual mosquito's, regarded in isolation, and then, knowing the relevant properties of the nodes, our assertions about the whole network. Intuitively I prefer the second approach, but I must admit that it is rather unclear to me, how such prrofs should look like. The article by W.H.Burge "Stream Processing Functions" (IBM J.Res.Develop., Vol.19, No. 1, pp 12 - 25) as yet does not strike me as very helpful.

7) How to design elephants.

8) etc.

It is nearly three weeks ago, since I started on this preliminary investigation and it seems that the time has come to sollicit comments.

5th March 1975
Plataanstraat 5
NUENEN - 4565
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow