About robustness and the like.

I cannot expect the following to exceed the status of a very tentative and preliminary report, for that is what it is as far as I am concerned. In the year behind me I have confined my attention --intentionally!-- almost exclusively to one-and-a-half aspect of the programming task. Let me summarize it, so that I can use it as my starting point.

The one side had to do with "program correctness". For this purpose I considered a program as a code for a predicate transformer that for each post-condition R allowed us in principle to derive the corresponding weakest pre-condition for the initial state that would guarantee a terminating process ending in a final state satisfying R . In order to get a balanced view of the significance of that theory, the following remarks seem in order.

1. For "crazy" R --i.e. a totally unintended post-condition-- the method works only "in principle", namely in the sense that carrying out the predicate transformation would lead to totally unmanageable formulae. I do not consider this as a serious defect -- as a matter of fact: as no defect at all.

2. Focussing attention upon the intended post-condition, the formalism could be used in a constructive manner, designing programs for which a sufficient pre-condition could be derived as well. The corresponding formal discipline is certainly on the plus-side of the achievement.

3. In order to get not too excited we should observe that the formal method referred to is quite sufficient when we want to compute the greatest common divisor, because there both problem and answer are phrased in terms of integers, i.e. the basic subject matter being manipulated by our programming language. Quite often, however --when dealing with graphs, for instance-- we introduce (because neither problem nor answer is really stated in terms of integers) a convention for representing the current state of affairs --initial, intermediate and final-- with the aid of integer variables and arrays. In one direction we introduce a representation, in the other direction an abstraction function, associating with each collection of integer values a unique value in the abstract domain. The question whether this representation/abstraction is adequately mirroring our intentions has largely been left outside our considerations. (To quote an older sentence, by which I tried to capture this state of affairs: "The only thing a computer can do is the manipulation of symbols, the only reason for doing so is that the symbols stand for something else.")

3.1. I think that I know what has to be done: the abstract values and the operations upon them have to be captured in a formal system. Independent of the program we can ask ourselves if we think that the axioms of that formal system capture the subject matter we would like to talk about. (I hope that this remark silences the complaint:"When do you know that the given specifications are the ones intended?". The obvious answer is:"Never, but we can do our best in trying to be pretty sure.") Using techniques --see Hoare's article on the correctness of data representations in ACTA INFORMATICA-- we can verify formally that our manipulations on our represented values satisfy the axioms of the abstract formal system. Usually we don't do that!

3.2. The situation is a little bit more serious in the sense that we are often willing to use known properties of the abstract world to "guarantee" things about the histories displayed by the represented values manipulated.

We shall argue that a certain loop does terminate because we can related each history to a directed path on a graph --our abstract entity-- of which we know that it has no cycles! And such a sweeping statement is not without danger when, for instance, each terminal "leaf" is represented as a node with an arc leading from itself to itself! Yet we do it, and I do as yet see no real alternative. The best seems to be aware of it --and to hesitate three or four times when introducing "a cunning representation convention". Without further information and further thought I assume for the time being that this oscillation between "the representation" and "what it stands for" is an intrinsic part of the programmer's game, of which he had better be aware! (It could be also this oscillation, which makes programming so difficult for people untrained to switching between levels.)

4. On the plus-side is certainly that our approach has shown the possibility to separate to a much higher degree than I was able to achieve before, the correctness concerns on the one hand from the efficiency concerns on the other. It is only for the latter concerns --which only have a meaning with respect to an implementation-- that the other interpretation of the program text --viz. as executable code-- becomes relevant.

The half aspect taken into account during the past year was "the corresponding computational processes". It was only taken into account as a source of efficiency considerations --based upon fairly neutral assumptions regarding time and space requirements of the implementation-- , as a means of providing a motivation for preferring one possible (correct) solution above another possible (and equally correct) one. Again some remarks are in order.

5. Under the assumption of availability of (in particular) the array operations --the fact that in actual fact integers stored have to remain within a bounded range can be dealt with in the usual way-- our mini-language allowed very efficient and delightfully compact algorithms of great sophistication, and was certainly helpful in their discovery. This was very encouraging and it was refreshing to get hints that many of the usual bells and whistles of our powerful programming languages --possibly including recursion!-- had perhaps better be discarded.

6. Thanks to the fact that precise execution times had been left undefined a host of optimization problems became meaningless (leaving one's mind free to replace an algorithm of cubic growth by one of quadratic growth, say). It is a mistake to think that without precise execution times, the programming task --at least the efficiency part of it-- becomes empty or trivial. When we shift our attention from "efficiency" to "robustness", we hope --and to be quite honest: trust-- to be able to do so without (in complete analogy) postulating precise probabilities for various forms of malfunctiong, yet retaining the ability to "increase robustness by an order of magnitude" in very much the same way as we could "increase the efficiency by an order of magnitude" without fixing the execution times.

6.1. The unwillingness to make precies assumptions about the execution times or fault rates is not only a (commendable!) lazyness from my side: it does also provide a means by which one's considerations gain in general applicability. (How many of Don Knuth's optimizations based on counting memory accesses will loose their validity under the assumption of a small associative store?)

6.2. My stubborn unwillingness to restrict the erratic behaviour of the daemon which is assumed in the implementation of non-determinacy is a decision of the same category.

As I have argued elsewhere --in EWD447-- scientific thought derives its effecitiveness from our willingness and ability to isolate an aspect of our problems temporarily and to try to study it for a while in isolation, for the sake of its own consistency, so to speak. This "focussing of one's attention" is different from (completely) ignoring the other aspects, for the one who does the latter is, indeed, a narrow-minded fool whose work cannot be expected to have any significance outside the little world he has created for himself. Consciously trying not to be a narrow-minded fool, I have tried to focus my attention upon the correctness problem, without forgetting about the engineering considerations that are related to the computational histories. There was a reason for tackling the correctness issue first --besides it being easier-- and that is the following. (I mention it, because it seems often overlooked.)

In very pragmatic environments, it is often argued that "our software need not be more reliable than our hardware". (Note. This statement only makes sense if, with respect to software, the adjective "reliable" can be used in exactly the same meaning as with respect to hardware --otherwise we let ourselves in for statements like "this cup of tea is sweater than that cup of coffee is hot"--; anyhow I have my doubts.) As an assumed consequence it is allowed that programs may contain errors producing erroneous results, erroneous results that could have been provoked by hardware malfunctioning as well: they will be caught by the same recovery mechanisms that have to be included anyhow. Here, however, I have a few comments.

7. From a point of view of maintenance of the hardware --of which we must assume, in contrast to software, that it is subject to wear and tear-- it is highly desirable to be allowed to conclude from certain recovery needs that only -hopefully rather specific-- hardware malfunctioning can have caused them!

8. Even if we allow malfunctioning and introduce recovery protocols, we would like the recovery process itself --please!-- to be correctly programmed! This gives a certain technical priority to the problem of program correctness over recovery.

9. The task of recovery is one that I can only understand in relation to what a correct program correctly executed would cause to happen. This gives a certain philosophical priority to the problem of program correctness.

10. I am not sure that such systems of hierarchical recovery do not suffer from the same disease as most "system engineering systems" seem to suffer from: having understood from Wiener that a system should have feedback, they make a religion out of "design iteration cycles" without worrying whether the iteration converges towards something acceptable. Looking at the history of OS-like projects, one can only get the impression that they often don't do so.

* * *

In the three-day interval denoted by the above stars I cleared up a confusion in my mind, that I must deal with first, before I can proceed. It has to do with so-called "defensive programming".

We teach students explicitly that if in a certain point in the program only three cases are allowed to occur, and all three have to be dealt with separately, the program should not test explicitly for the presence of two cases, and treat by default anything else as "the third case", but that for the third case should be tested explicitly as well. (That with the advent of the guarded commands the temptation for which we had to warn has largely disappeared, is fine, but here of no relevance.) In the history of the program, including its development, such "superfluous tests" played, however, very different roles!

In the beginning --we know how programs used to be!-- they were a debugging aid. And upon the alarm that a fourth case, uncatered for, was presented, we usually still had to figure out, whether not catering for the fourth case had been an omission of that program part, or whether the fourth case had been erroneously produced by a program bug somewhere else! (It is typically the function of explicit program interfaces to settle that dispute in advance.) In any case the alarm was taken at the beginning as the indication of a program bug. By the time that one was fully confident in the program's correctness --fully confident to the standards of those days-- , however, one did not remove the checks, on the contrary! If, after intensive use of such a debugged program, one of these checks suddenly gave an alarm, the first thought was a detected machine malfunctioning.

If in the following I consider the justification for additional checks, it will always be in that second function. While designing "correct programs" as I have been considering during the past period, I always assumed a perfect machine in the case that you wanted the program executed. (Perhaps you were not interested in its execution at all: the question of the program's correctness still makes sense.) But now I am shifting my attention from the program towards it execution, and now I must assume a perfect program! I pretend to be no longer interested in the question of the correctness of the program, but only to be interested in the correctness of the answer, the correctness of the execution. I know that this violates our common sense of modesty. Yet this is the only sensible assumption that I can make if I want to separate the various concerns, and if I don't do that, I know that I shall never come to grips with the problem. (If it hurts too much, we shall assume our programs to have been made by The Good Lord Himself, in exactly the same way as we have delegated last year the execution of our programs to The Good Lord's Machine GLM!)

* * *

So we envisage the situation of a perfect program executed by a possibly lousy machine, i.e. a machine that possibly does not provide a perfect implementation of our programming language (and those who regard this situation so utterly unrealistic as to have difficulty in reading on, I can only beg to have patience, lots of patience.....).

In our aims we may be modest or ambitious: in the modest approach we only try to decrease the probability of producing a wrong result, in the ambitious approach we try also to increase the probability of producing the right result, the difference being in the probability that the machine "gives up". In the modest approach the adagium will be "When in doubt, abstain!", in the ambitious approach the adagium will be "When in doubt, try something else, try to recover!". The ambitious approach is clearly self-defeating if the decreasing probability of "giving up" is bought at the price of an increasing probability of producing a wrong result, i.e. our modest approach

points at a goal that we should never forsake; it is the most fundamental of the two, and, therefore, I shall focus my attention upon that one first. (It has the added advantage of seeming to be the easier of the two.)

How can we increase our confidence --because that is now what it boils down to-- that during program execution nothing has gone wrong? Well, the machine can certainly assist by its very structure to increasing that confidence There is one very important way in which it can do so, so important as a matter of fact, that we may state that a well-designed machine must do that.

I assume that our programs have been written for the GLM, because that is a machine we can hope to be able to program for. The designer of the actual machine knows --or at least, he should know-- that he is not the Good Lord Himself, and that he can hope at most to build a partial simulator of the GLM. While in the GLM, for instance, there is in principle no upper bound on the maximum value of integer variables, the actual machine simulating the behaviour of the GLM may be such --usually is such-- that it can only cope with integers up to a certain limit. The simulator should check constantly whether it fails, not by virtue of malfunctioning, but by virtue of its designed construction, to simulate the GLM faithfully. As a result a test on overflow of integer capacity is absolutely essential and a machine which in order to remain in range, reduces integer values,for the sake of its own convenience and without warning, modulo something, is a monstrum, unfit for human use. From now onwards we assume that the design of the actual machine is perfect as well, perfect in the sense that -apart from malfunctioning-- no wrong answers will be produced in account of undetected inability to simulate the GLM, as incorporated in the design. (The simulation of the GLM's behaviour is only claimed for a subset of the computations that could take place in the GLM. The simulation as designed is only a partial function of the correct programs+input, and it is the duty of the actual machine to check that in this sense it is not invoked outside its domain.)

So far so good. But now the problems come. There are two types of results: there is the result that is laborious to find, but easy to check, once you have it, there is also the result that is not only laborious to find, but equally laborious to check, once you have it.

Suppose that we know how to multiply quickly, but don't know how to extract a square root. Than "finding the square root" will be regarded as a laborious process, in order to check it, we only need to compare the square of the result with the argument.

Suppose that we want a very large number to be factorized in prime factors. If the result is a long series of small factors of which we know that they are all prime numbers, we only need to multiply them with each other, in order to check that the original number returns. But what, if the outcome of the computation is that the given number itself is already a prime number? (It is then clearly a prime not known to us, because otherwise there would have been no point in providing it as the argument to our factorization program!)

At first sight, there seem only two ways of increasing our confidence in the correctness of a result that is as laborious to check than it is to find. And in a certain sense they both seem to double the costs of computation.

The one way is usually described as "repeating the computation". If the repetition is done --as they sometimes do-- with a different program and/or a different machine and/or a different mathematical approach, and the two answers confirm each other, we have checked in some way a lot more than the correct execution of a program. We shall --in accordance with the position taken-- ignore those additional advantages, and only remark, that comparing two independently derived results is only any good if the result is unique, if we could come away with a deterministic machine. I think --but this is no more than a feeling-- that even for the answer of a non-deterministic computation matters can be arranged in such a way that verification can be done at a price similar to construction of the result.

The other way is indeed relying on the result, "without back-substitution" so to speak, because one knows that the individual steps of the simulation of the GLM have been checked rather abundantly. The second approach has the undoubted advantage of being a general purpose solution; add to this the advantage of getting diagnostic information about hardware functioning, and it is clear that no computer manufacturer can afford not to explore the possibilities of that approach. It is, however, not the whole story, for such a machine makes the outcome of a long computation still less trustworthy than the outcome of a short one. Trying to supply the rest of the story is one of the things I should do!

* * *

The above, which has the nature of a research proposal, was written about three months ago, from which it can be deduced that, in the mean time, I have been engaged on other tasks. At odd moments I have given some attention to the problem of increasing the reliability of answers produced by a not fully reliable machine, but I intend to describe these exercises in separate documents. I can, however, already mention one tentative conclusion.

One way of trying to prevent the machine from producing a wrong answer is trying to prevent it from making <u>any</u> undetected error. This very puritan attitude shifts the stress from the correct answer to the correct machine, and from the point of view of hardware maintenance, it might be the most helpful one.

My experiments, however, seemed to indicate that "checking the machine" to such an extent is a very hard problem, and that the whole problem becomes more manageable if we don't care for such machine malfunctions that, although "malfunctions" in the sense of not being intended behaviour, are harmless insofar that, despite their occurrence, the final answer will still be correct.

The simplest example of such a malfunctioning is when in a repetitive construct

<u>do</u> B → S <u>od</u>

after B has been correctly evaluated to the value "true" the execution of S erroneously reduces to the empty statement "skip". This is a completely harmless error, and I am afraid that it would be very expensive to catch.

Although being a puritan by nature, I expect therefore to confine my attention at first to the prevention of the generation of wrong results.

31st January 1975

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow