A preliminary investigation into Computer Assisted Programming

This is a very tentative document on an effort to increase our programming ability.

I feel that this problem will become very urgent: the programmer finds his task in the field of tension between the available machines and the computations we want to have performed by them. As available machines become more and more powerful, mankind will become more and more ambitious in their applications and programs will grow in size and complexity. It takes no great prophet to forecast that in the years to come the mechanical execution of the program once it is there will be the minor problem, whereas the major problem will be the process of program composition itself.

I accept that the intellectual effort needed to compose a program (measured in some loose sense) is an increasing function of program length (measured in an equally loose sense). The point is to what functional dependence we are aiming! I accept a programming technique such that the effort needed is proportional to the length. As long as we only have a programming technique such that the effort needed is proportional to, say, length squared, we had better admit defeat. (I am afraid current programming techniques are often even worse than that: sometimes I suspect an exponential growth of the effort needed!)

Over the past ten years we have witnessed an explosive increase of our programming ability as embodied by the advent of the so-called "higher level programming languages". Yet I have the feeling that they have done more to decrease some rates of proportionality than to change the functional dependence itself. We all know that the use of higher level programming languages made the construction of larger programs feasible; but we also know that at a certain stage a program written in a higher level programming language becomes just as intricate as the machine code programs of ten years ago. When one hits ones head at the ceiling it is a meagre consolation to know that it was at the ceiling of the next floor: the pain is just the same!

In the past year I have done a number of experiments. They were non-mechanical in the sense that I still viewed the programmer's task as producing a handwritten program text. In other words: I have been looking for a mental discipline. As yet, I do not regard these investigations as closed.

Among other things I have tried to behave, consciously, as "the ideal programmer" as I saw him at that moment. Without recording the experiments in detail now, I should like to give the core of some observations made.

First, the effort has been non-empty in the sense that I am appalled now by programs I made only three years ago. (This is a very strong observation, for among the appalling programs is the first one occurring for illuminating purposes(!) in an introductory(!) course on "Algorithms". At the time of writing -I remember quite well- I was delighted by its clarity!)

Second, the experiments have convinced me that "first making the program and then debugging it" is like putting tha cart before the horse. The more I think about "debugging" in this sense, the more hopeless it becomes; in actual fact it now strikes me as a dead alley. And I am now much more in favour of a constructive approach to the problem of program correctness, i.e. to control the process of program composition in such a way as to prevent bugs from entering the construction.

I know that this goal strikes many as impossible to achieve, particularly those who never tried it seriously. I will not deny that it is hard work -making a large and complicated program will be hard work anyhow!-; my claim is that when one makes a conscious effort at it, it is more feasible than trying to get a program correct when the problem of correctness has been left to the debugging phase.

Third, I have observed that by the time the program was completed, so was its "documentation". Here I use the term "documentation" in a completely informal sense, as a guide to understanding the program. Approaching the program through this documentation is reading the proof of its correctness.

Finally, the observation that even in the construction of very small programs the production of this documentation (or should I say "prae-documentation" to drive the message home?) is already a very rewarding exercise. In the last year I have taken oral examinations of the course "Introduction into the Art of Programming". (The oral examinations took more of my time than the actual lecturing, but as far as I am concerned the time has been well-spent. I got the feedback necessary to improve the lectures; above that it gave me an opportunity for direct observation of programmer's misbehaviour!) Lately, a student left open the boolean expression controlling a repetition clause and he had not made the prae-documentation explicit, neither in writing nor in his mind; when he had to fill in the correct boolean expression I saw him reading, statement after statement, the interior of the clause five(!) times and then he filled in the boolean expression (erroneously, by the way), whereas two lines of reasoning would have been sufficient. It was a most convincing affirmation of my fear that many a programmer looses much time and energy in trying to read the meaning of what he has written.

Summarizing, I can state without exaggeration that a very promising discipline is emerging; even when the project of Computer Assisted Programming does not materialize, this line of thought will be pursued.

The purpose of this paper is to raise the question whether it is sufficient to regard documentation purely as a human guide to the program or whether we must try to regard it as an essential part of it. It is impossible to give a motivated answer to the question "How" before we have an idea of the kind of benefits we hope to derive from it.

Why is it so hard, at present, to force programmers to make the appropriate prae-documentation? There is a historical reason: they have not been trained to do so. There is a psychological reason: they overestimate their powers and think that they can do without it and experience the making of prae-documentation as an additional burden. There is a practical reason, justifying this latter point of view: whatever documentation they prepare about the program they are making, it remains a guide for <u>humans</u> and it cannot be mechanically used in the actual composition of the program (unless one regards an ALGOL-text as "documentation" about the corresponding object program).

The next remark is related to the fact that every large program will have to exist in a number of versions. (A first version may be logically correct, but its performance may be unsatisfactory; above that the demands made upon it wil vary during its lifetime.)

The naive approach to this situation is that we must be able to modify an existing program (program maintenance); the task to be performed is then presented as one of text manipulation.

I would like to approach this problem from another angle and would like to treat the task of program composition and that of program modification as essentially the same.

In the process of program composition many "small" decisions have to be taken: some of these decisions can be taken in parallel (i.e. independent of each other), other decisions must be taken one after the other (i.e. it is the one decision that makes the other relevant). It is certainly the task of the prae-documentation to record the intermediate stages.

If a program has to exist in two versions I do not like to regard (the text of) the one as a modification of (the text of) the other, I should like to relate them both to their common ancestor that (hopefully!) occurs already in the prae-documentation. The first intention is that the two versions share their respective correctness proofs as much as possible; the second intention is that they share mechanically as much as possible of the common (or "equal") coding.

In making a program I want to regard the target program no longer as an isolated object, I wish to treat it as a member of a structured class of similar programs (either alternative programs for the same task or different programs for similar tasks). And I wish to express this explicitly.

At first sight this may seem to impose a considerable additional burden on the shoulders of the poor programmer: instead of requiring the construction of a single program we require at least the conception of a whole class of programs. But I am not so sure. If you have to prove that the three perpendiculars of a given triangle pass through one point, you prove it for any triangle! That is what abstraction does for you and abstraction permeates the whole subject. To have a specific computation performed one writes an algorithm that could perform a whole class of computations and one proves theorems about the whole class although finally perhaps only one of the class will be performed!

Similar remarks apply to the design process itself, which by its (or "our"!) very nature is a sequential one. At a certain intermediate stage one does not have an incomplete program that has to be completed to make sense, one has a program that at a certain level of detail (or, viewing it from the other side, "at a certain level of abstraction") is complete and correct. As such one has abstracted from the decisions that are still to be made!

Much of the programmer's wisdom will be reflected in the choice of programs he includes in the class considered. As said before: later program modifications should be anticipated already in the prae-documentation which should already contain a (hopefully) close ancestor for the next version. This is already now a vital part of the programmer's duty; a system of Computer Assisted Programming may act as a reminder to this obligation.

I must mention two other considerations that suggest that some form of documentation about the program should be regarded as an integral part of it. At present we feed the machine with programs and this is the bare minimum, because the machine can execute a program "without understanding it". But as long as only the bare program and nothing else is inside the machine we are faced with two problems -already urgent now- that are pretty hopeless.

The one problem is the design of a proper reaction to a detected malfunctioning of a piece of selfchecking machinery -parity check, say. We have no scale along which to measure the size of the calamity: in our system the effect of a machine error may be confined to a single program or the system as a whole may derail. I feel that the present absence of a grip on the structure of the computations is one of the main causes of the current failure to construct dual systems for larger safety.

The second problem -one that attracts me much more than partial fault recovery- is to modify a program while it is working. Although the need is obvious we don't even have the concepts in terms of which the problem can be clearly stated.

To keep in mind that in future the computer itself may play an active (or to use the present OK terminology: "interactive") role in the process of program composition has the following motivation. If it does not materialize, it will nevertheless be a constant reminder to look for a formal method to give the prae-documentation. Being rather verbal such a constant reminder will be very useful. If it does materialize, there are many potential benefits, apart from the final product.

To start with, the system itself will be "a large, complicated program" and it should assist in its own construction. A kind of logical bootstrapping. The construction of the system itself will be the first testcase of the methods! A next consideration is that using the system to construct itself will provide in a nut shell the laboratory conditions for a program to be changed while it is in action!

Finally, why should I try to do it? I think it safe to say that the relevance of the project is beyond question. Then two questions remain: is no one else already doing it and am I sufficiently equipped to try it? To get some kind of answer to the first part of the question I have pumped Brian Randell but as far as he knew what he had seen of my approach seemed to him rather unique. Industrial concern seems to center on a PERT-like speeding up of a specific design and on the timely discovery (by simulation) where bottlenecks are to be expected in the performance more than on a classification of the possible designs on account of the degrees of freedom. Correctness proofs are certainly "in the air" (McCarthy, Naur, Floyd, Hoare in "The Axiomatic Method"), to use them as guiding principle in the process of program composition I have only seen by Floyd ("Assigning Meanings to Programs"), but as far as I know, Floyd has no actual experience in designing large programs.

This leaves the question "Why I?". I have, I think a claim to priority. I quote myself (1962) "In particular I would require of a programming language that it should facilitate the work of the programmer as much as possible, especially in the most difficult aspects of his task, such as creating confidence in the correctness of his program. This is already difficult in the case of a specific program that must produce a finite set of results. But then the programmer only has to show (afterwards) that if there were any flaws in his program they apparently did not matter (e.g. when the converging of his program is not guaranteed beforehand). The duty of verification becomes much more difficult once the programmer sets himself the task of constructing algorithms with the pretence of general applicability.

But the publication of such algorithms is precisely one of the important fields of application for a generally accepted machine independent programming language. In this connection, the dubious quality of many of the ALGOL 60 algorithms published so far is a warning not to be ignored."

Then came the multiprogramming system in the asynchronous part of which the design has been heavily influenced by correctness concerns; and this to our great advantage. In its connection three remarks are in order. First that the very high fame/publicity ratio can be taken as an indication that such an approach is (still) fairly exceptional. Second, that although this was not our primary aim, the approach brought with it in a very natural fashion a fair amount of modularity (whatever that may mean exactly). Finally, and this is now very important, that the final product shows where we have failed: the last -I would like to say: "extensive"- production phase has been too primitive and the resulting system is monolithic. It is an unmanageable, unmodifiable program, not in principle but for the labour involved. I feel that this is mainly the result of our manual production technique. It is, I think, also caused by the fact that at the end our self-discipline failed and we did not stick to our principles: at the lowest level -and there is so much lowest level!- we just programmed.

To the question "Why I?" I am inclined to answer the following. It is the natural consequence of my concerns over the last six or more years, extensive experiences gained are relevant and, reviewing them, I am beginning to get a feeling where I succeeded and where I failed, where and why.

Two final remarks about the possible scope of the project. In my more optimistic moments I do not exclude the possibility that the techniques will evolve to such a point that I can transform in a number of well understood steps an interpreter for an algebraic language into a compiler for that same language. If this turns out to be possible, the bootstrapping technique becomes more realistic. The second remark is that I may make the impression of reinventing flowdiagrams. Perhaps this impression is correct; there may be a point in doing so! The technique of flowdiagrams has been conceived at a time that programs were several orders of magnitude smaller than the programs we have to make now and may very well be in need of revision.