The complete system consists of a system tape and some associated documentation. Manuals will be provided to describe the definition, implementation and use of the language. Another manual will discuss the procedures for installing, improving and maintaining the system.

The system tape consists of six logical files. The first contains the ALTRAN character set and the second contains an editor to be used for incorporating system updates. The third file contains the macro processor, M6 in source language and the fourth contains some basic macro definitions. Finally, the last two files contain the FORTRAN and ALTRAN parts of the system, respectively.

To install the system (see Figure 2), a recipient must first write some basic macros and some primitives, and transliterate the system tape into his local character set. Next he compiles the macro processor using his local FORTRAN compiler. Then he compiles the translator, the interpreter, and the FORTRAN part of the library, using the macro processor followed by the FORTRAN compiler. Then he compiles the ALTRAN part of the library, using the ALTRAN translator and the FORTRAN compiler. Finally he compiles and runs an ALTRAN test program to be sure that everything is in working order.

Once the system is operational, it can be improved by replacing critical procedures with more efficient machine language versions. These replacements can be made at leisure and in an order determined by local experience. However, the documentation indicates procedures that are known to be critical.

Improvements will be distributed occasionally in the form of an update letter. The changes can be keypunched from the update letter and incorporated with the aid of the editor. The editing technique has been designed in such a way that unrelated local modifications need not interfere with this process.

84

References

1. **Brown, W. S.** *The ALTRAN language for symbolic algebra,* unpublished, January 1969, pp. 84.

2. **Hall, A. D.** *The M^6 macro processor;* (unpublished), March 1969, pp. 16, 1 appendix. M6 was designed by M. D. McIlroy, using ideas from many sources. The FORTRAN implementation is a translation by A. D. Hall of an original written in MAD by R. Morris.

7.4 **Structured programming**

by

E. W. Dijkstra

Introduction

This working document reports on experience and insights gained in programming experiments performed by the author in the last year. The leading question was if it was conceivable to increase our programming ability by an order of magnitude and what techniques (mental, organizational or mechanical) could be applied in the process of program composition to produce this increase. The programming experiments were undertaken to shed light upon these questions.

Program size

My real concern is with intrinsically large programs. By “intrinsically large” I mean programs that are large due to the complexity of their task, in contrast to programs that have exploded (by inadequacy of the equipment, unhappy decisions, poor understanding of the problem, etc.). The fact that, for practical reasons, my experiments had thus far to be carried out with rather small programs did present a serious difficulty; I have tried to overcome this by treating problems of size explicitly and by trying to find their consequences as much as possible by analysis, inspection and reflection rather than by (as yet too expensive) experiments.

In doing so I found a number of subgoals that, apparently, we have to learn to achieve (if we don’t already know how to do that).

If a large program is a composition of N "program components", the confidence level of the individual components must be exceptionally high if N is very large. If the individual components can be made with the probability "p" of being correct, the probability that the whole program functions properly will not exceed

$$P=P^N$$

for large N, p must be practically equal to one if P is to differ significantly from zero. Combining subsets into larger components from which then the whole program is composed, presents no remedy:

$$p^{N/2} * p^{N/2} \text{ still equals } p^N \text{ !}$$

As a consequence, the problem of program correctness (confidence level) was one of my primary concerns.

The effort—be it intellectual or experimental—needed to demonstrate the correctness of a program in a sufficiently convincing manner may (measured in some loose sense) not grow more rapidly than in proportion to the program length (measured in an equally loose sense). If, for instance, the 85 labour involved in verifying the correct composition of a whole program out of N program components (each of them individually assumed to be correct) still grows exponentially with N. we had better admit defeat. Any large program will exist during its life-time in a multitude of different versions, so that in composing a large program we are not so much concerned with a single program, but with a whole family of related programs, containing alternative programs for the same job and/or similar programs for similar jobs. A program therefore should be conceived and understood as a member of a family; it should be so structured out of components that various members of this family, sharing components, do not only share the correctness demonstration of the shared components but also of the shared substructure.

Program correctness

An assertion of program correctness is an assertion about the net effects of the computations that may be evoked by this program. Investigating how such assertions can be justified, I came to the following conclusions:

1 The number of different inputs, i.e. the number of different computations for which the assertions claim to hold is so fantastically high that demonstration of correctness by sampling is completely out of the question. *Program testing can be used to show the presence of bugs, but never to show their absence!* Therefore, proof of program correctness should depend only upon the program text.

2 A number of people have shown that program correctness can be proved. Highly formal correctness proofs have been given; also correctness proofs have been given for "normal programs", i.e. programs not written with a proof procedure in mind. As is to be expected (and nobody is to be blamed for that) the circulating examples are concerned with rather small programs and, unless measures are taken, the amount of labour involved in proving might well (will) explode with program size.

3 Therefore, I have not focused my attention on the question "how do we prove the correctness of a given program?" but on the questions "for what program structures can we give correctness proofs without undue labour, even if the programs get large?" and, as a sequel, "how do we make, for a given task, such a well-structured program?". My willingness to confine my attention to such "well-structured programs" (as a subset of the set of all possible programs) is based on my belief that we can find such a well structured subset satisfying our programming needs, i.e. that for each programmable task this subset contains enough realistic programs.

4 This, what I call "constructive approach to the problem of program correctness", can be taken a step further. It is not restricted to general considerations as to what program structures are attractive from the point of view of provability; in a number of specific, very difficult programming tasks I have finally succeeded in constructing a program by analyzing how a proof could be given that a class of computations would satisfy certain requirements; from the requirements of the proof the program followed.

The relation between program and computation

Investigating how assertions about the possible computations (evolving in time) can be made on account of the static program text, I have concluded that adherence to rigid sequencing disciplines is essential, so as to allow step-wise abstraction from the possibly different routing. In particular: when programs for a sequential computer are expressed as a linear sequence of basic symbols of a programming language, sequencing should be 86 controlled by alternative, conditional and repetitive clauses and procedure calls, rather than by statements transferring control to labelled points.

The need for step-wise abstraction from local sequencing is perhaps most convincingly shown by the following demonstration:

Let us consider a “stretched” program of the form

$$S_1; S_2; \ldots; S_N \qquad (1)$$

and let us introduce the measuring convention that when the net effect of the execution of each individual statement S_j has been given, it takes N steps of reasoning to establish the correctness of program (1), i.e. to establish that the cumulative net effect of the N actions in succession satisfies the requirements imposed upon the computations evoked by program (1).

For a statement of the form

$$\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \qquad (2)$$

where, again, the net effect of the execution of the constituent statements S_1 and S_2 has been given; we introduce the measuring convention that it takes 2 steps of reasoning to establish the net effect of program (2), viz. one for the case B and one for the case not B.

Consider now a program of the form

```
if B_1 then S_11 else S_12;
if B_2 then S_21 else S_22;
        .
        .
        .
if B_N then S_N1 else S_N2                    (3)
```

According to the measuring convention it takes 2 steps per alternative statement to understand it, i.e. to establish that the net effect of

$$\textbf{if } B_i \textbf{ then } S_{i1} \textbf{ else } S_{i2}$$

is equivalent to that of the execution of an abstract statement S_1. Having N such alternative statements, it takes us 2N steps to reduce program(3) to one of the form of program (1); to understand the latter form of the program takes us another N steps, giving 3N steps in toto.

If we had refused to introduce the abstract statements S_i but had tried to understand program (3) directly in terms of executions of the statements S_{ij}, each such computation would be the cumulative effect of N such statement executions and would as such require N steps to understand it. Trying to understand the algorithm in terms of the S_{ij} implies that we have to distinguish between 2N different routings through the program and this would lead to N*2N steps of reasoning!

I trust that the above calculation convincingly demonstrates the need for the introduction of the abstract statements S_i. An aspect of my constructive approach is not to reduce a given program (3) to an abstract program (1), but to start with the latter.

Abstract data structures

Understanding a program composed from a modest number of abstract statements again becomes an exploding task if the definition of the net effect of the constituent statements is sufficiently unwieldy. This can be overcome by the introduction of suitable abstract data structures. The situation is greatly analogous to the way in which we can understand an ALGOL program operating on integers without having to bother about the number representation of the implementation used. The only difference is that now the programmer must invent his own concepts (analogous to the "ready-made" integer) and his own operations upon them (analogous to the "ready-made" arithmetic operations).

87

In the refinement of an abstract program (i.e. composed from abstract statements operating on abstract data structures) we observe the phenomenon of "joint refinement". For abstract data structures of a given type a certain representation is chosen in terms of new (perhaps still rather abstract) data structures. The immediate consequence of this design decision is that the abstract statements operating upon the original abstract data structure have to be redefined in terms of algorithmic refinements operating upon the new data structures in terms of which it was decided to represent the original abstract data structure. Such a joint refinement of data structure and associated statements should be an isolated unit of the program text: it embodies the immediate consequences of an (independent) design decision and is as such the natural unit of interchange for program modification. It is an example of what I have grown into calling "a pearl".

Programs as necklaces strung from pearls

I have grown to regard a program as an ordered set of pearls, a "necklace". The top pearl describes the program in its most abstract form, in all lower pearls one or more concepts used above are explained (refined) in terms of concepts to be explained (refined) in pearls below it, while the bottom pearl eventually explains what still has to be explained in terms of a standard interface (=machine). The pearl seems to be a natural program module.

As each pearl embodies a specific design decision (or, as the case may be, a specific aspect of the original problem statement) it is the natural unit of interchange in program modification (or, as the case may be, program adaptation to a change in problem statement).

Pearls and necklace give a clear status to an "incomplete program", consisting of the top half of a necklace; it can be regarded as a complete program to be executed by a suitable machine (of which the bottom half of the necklace gives a feasible implementation). As such, the correctness of the upper half of the necklace can be established regardless of the choice of the bottom half.

Between two successive pearls we can make a "cut", which is a manual for a machine provided by the part of the necklace below the cut and used by the program represented by the part of the necklace above the cut. This manual serves as an interface between the two parts of the necklace. We feel this form of interface more helpful than regarding data representation as an interface between operations, in particular more helpful towards ensuring the combinatorial freedom required for program adaptation.

The combinatorial freedom just mentioned seems to be the only way in which we can make a program as part of a family or "in many (potential) versions" without the labour involved increasing proportional to the number of members of the family. The family becomes the set of those selections from a given collection of pearls that can be strung into a fitting necklace.

Concluding remarks

Pearls in a necklace have a strict logical order, say "from top to bottom". I would like to stress that this order may be radically different from the order (in time) in which they are designed.

Pearls have emerged as program modules when I tried to map upon each other as completely as possible, the numerous members of a class of related programs. The abstraction process involved in this mapping turns out (not, amazingly, as an afterthought!) to be the same as the one that can be used to reduce the amount of intellectual labour involved in correctness proofs. This is very encouraging.

As I said before, the programming experiments have been carried out 88 with relatively small programs. Although, personally, I firmly believe that they show the way towards more reliable composition of really large programs, I should like to stress that as yet I have no experimental evidence for this. The experimental evidence gained so far shows an increasing ability to compose programs of the size I tried. Although I tried to do it, I feel that I have given but little recognition to the requirements of program development such as is needed when one wishes to employ a large crowd; I have no experience with the Chinese Army approach, nor am I convinced of its virtues.