A CASE STUDY OF STRUCTURED PROGRAMMING WITH CORRECTNESS PROOFS

J.D. Ichbiah, J.C. Heliard

Compagnie Internationale pour l'Informatique, Louveciennes, FRANCE

1. INTRODUCTION : STATIC PROGRAMMING

One of the starting points of Structured Programming is the recognition of the fact that our mind is "rather geared to master static relations" than dynamic, time evolving relations {1, 2}.

A well-structured program is thus a program which is understandable in a static fashion. It must be possible to qualify each point of such a program by a set of properties, called invariants, which are satisfied whenever this point is reached, and for arbitrary executions of the program.

The main justification for the simplicity of the control structures used in structured programming (sequences of statements, if and case statements, simple loops, subprograms) is precisely that they simplify the deduction of the properties satisfied at a given program point from those satisfied at preceding points.

Similarly, the systematic use of typed data contributes to the static character of structured programs. Thus the declaration

```
x : (green|orange|red) ;
```

introduces an invariant relation for all the life-time of the variable x, that is, the value of x is either "green" or "orange" or "red", all other values being excluded.

This same character of static programming is also found in program verification theory {3, 4, 8}. In order to explicit this static character let us review the principles of these verifications.

To verify a program Q with respect to a precondition P and a resulting condition R is to prove that if P holds before execution of Q then the condition R will hold after execution of Q. We write this P{Q}R in Hoare's notation.

The verification of a program which is a sequence of statements {Q1 ; Q2} may be sub-divided into a sequence of verifications. Thus, in order to prove P{Q1 ; Q2}R it suffices to prove that P{Q1}R1 and R1{Q2}R. In other words, it must be shown that the condition satisfied after execution of Q1 is the precondition for the execution of Q2 to produce R.

This work has been supported by Iria-Sesori.

Similarly, the verification of a conditional statement may be broken down into elementary verifications. Thus in order to show that P{IF B THEN Q1 ; ELSE Q2 ; END}R we must show that P AND B {Q1} R and also that P AND NOT B {Q2} R.

The verification of a loop P {UNTIL B LOOP Q ; REPEAT} R necessitates the identification of a condition I which is implied by P and which is such that I AND B implies R. Furthermore, I must satisfy I AND NOT B {Q} I. In other words, I must be invariant through the execution of the loop.

To summarize, the verification of a program reduces to that of a set of elementary conditions which do not depend on specific executions of the program. As in structured programming, all dynamic aspects have been abstracted from the verification.

Nevertheless a major difficulty remains which is the determination of loop invariants. In general, this requires a deep understanding of the programs considered. Consequently, a posteriori certification appears impractical for programs written initially without any intent of proof. Even if it demonstrates the feasibility of this approach for programs of up to a few hundred lines, the work done by London and his followers {6} does not enable one to conclude that the approach can be generalized economically.

Hoare {5}, was the first to propose the integration of verifications in the construction process of structured programs. The previous discussion, which has brought out the common static character of both structured programming and program verifications, shows that this is a very natural idea. Indeed it has been followed by several authors {7, 9, 10, 11}.

We have experimented with this approach during the construction of a program of more than 2000 lines, written in LIS {12} and treating a non-trivial problem.

As an example, the proof performed for one of the sub-programs of this program is given in detail in the second part of this paper. It should be noted that this is the proof of a sub-program dealing with structured data and pointers.

Finally we conclude by a global appraisal of this experiment of structured programming with informal program proofs.

2. EXAMPLE : SORTING A LINEAR LIST

The example described hereafter involves linear lists of qualified elements. The problem is to write a program which reorders the elements of a list in such a way that elements having the same qualification appear consecutively in the final list. As an additional requirement, the order in which elements of a same qualification appear may not be modified.

On the average, the lists considered are rather short (say ten elements) but much longer lists will also occur. Qualifications are analogous to names ; there is an infinite number of possible qualifications. In most cases there will be very few elements having the same qualification, if any.

Clearly, a classical sorting algorithm could be used for the problem considered. However, the short length of the lists does not justify the use of sophisticated algorithms. In addition, the need to decode and encode the list information in a form acceptable for a given sorting routine would offsett the advantage of having an already written algorithm. Hence a sorting algorithm working directly on the list structure will be developed. The emphasis will be on correctness, not on time considerations.

The precondition P is expressed in the form "F-L is a list". It means that :

P : "F-L is a list"

(a) F designates the first element
(b) L designates the last element
(c) For each element X,
    X.QUAL designates the qualification of X
    X.SUC designates the successor of X
(d) L.SUC = NIL
(e) The elements of the list are
    F, F.SUC, F.SUC.SUC, ..., L

Note that the "successor" relation (direct or indirect) is an order relation on the elements of the linear list. By convention we will write A <= B to indicate that B designates either a successor of A or the same element as A.

With this notation, the condition R to be satisfied at the end of the program is :

R

(1) F-L is a list
(2) If A and B satisfy F <= A <= B <= L and
    if A.QUAL = B.QUAL then
    (a) A <= B was also true initially
    (b) For each element X such that A <= X <= B,
        we have A.QUAL = X.QUAL = B.QUAL

The precondition P and the resulting condition R form the specification of our program at level zero. Proceeding top-down, we will successively introduce program specifications at lower levels.

The proof issued at a given level may require that a condition be satisfied by a lower level. Such conditions will be indicated by a vertical bar in the left margin.

LEVEL 1

The principle of an iterative solution may be derived from the statement of R. To that end we introduce an intermediate element G subject to a condition (the G invariant) similar to R :

I(G)

If A and B satisfy F <= A <= B <= G and
if A.QUAL = B.QUAL then
(a) A <= B was also true initially
(b) for each element X such that A <= X <= B,
we have A.QUAL = X.QUAL = B.QUAL

Hence the first statement of the algorithm will be :

ALGORITHM A1

```
G := F ;
UNTIL G = L LOOP
   "advance G while maintaining I(G)" ;
REPEAT ;
```

PROOF 1

The condition I(G) is satisfied after the assignment "G := F". In addition, I(G) implies R at the end of the loop since G = L.

The program is hence correct provided that the loop terminates.

Note This formulation takes care of the cases

F = L = NIL (no element)
F = L ≠ NIL (a unique element)

LEVEL 2 "Advance G while maintaining I(G)"

In order to advance G we are looking for elements having the same qualification as G among its successors. If the immediate successor of G is a synonym, G is made to designate this successor, otherwise it is convenient to define FA, the first antonym of G :

I(FA)	(a) FA is the successor of G (FA = G.SUC) (b) FA is an antonym of G (FA.QUAL ≠ G.QUAL)

Once FA is located, me may remove the synonyms of G which are successors of FA and start another iteration with the successor of G (i.e. FA).

ALGORITHM A2

```
IF G.QUAL ≠ G.SUC.QUAL THEN
   FA := G.SUC ;
   "Remove synonyms of G which are successors of FA" ;
END ;
G := G.SUC ;
```

PROOF 2

. Within the loop G ≠ L implies G.SUC ≠ NIL. Hence G.SUC.QUAL has a meaning.

. If G.QUAL = G.SUC.QUAL, the invariant I(G) remains satisfied after the assignment G := G.SUC

If G.QUAL ≠ G.SUC.QUAL then for A2 to satisfy I(G) after the assignment G := G.SUC it suffices that the execution of "Remove ... FA" leave no synonym of G beyond FA and maintain I(G) and I(FA).

. The loop A1 terminates because of the assignment G := G.SUC (assuming also that elements of the sublist F-G are never added to the sublist G-L).

LEVEL 3 "Remove synonyms of G which are successors of FA"

We are looking for the synonyms of G which are between FA and L. To that end we introduce a cursor C defined as follows :

I(C)

(a) C is an antonym of G (C.QUAL ≠ G.QUAL)

(b) Every element X such that FA <= X <= C
is an antonym of G (X.QUAL ≠ G.QUAL)

ALGORITHM A3

```
C := FA ;

UNTIL C = L LOOP

   "Treat the successor of C" ;

REPEAT ;
```

PROOF 3

The invariant I(C) is satisfied after the initialization C := FA.

At the end of the loop C = L implies that all synonyms of G have been removed from FA-L provided that "Treat the successor of C" maintain I(C) and also I(G) and I(FA). In addition, termination of the loop A3 must be demonstrated.

LEVEL 4 "Treat the successor of C"

If the successor of C is not a synonym of G we simply let C progress down the linear list and designate this successor. Otherwise let us denote the successor of G by SG (Synonym of G).

We now have to extract the element designated by SG from the sublist C-L and to insert it between G and FA. Then SG may be taken as the new G.

ALGORITHM A4

```
IF G.QUAL ≠ C.SUC.QUAL THEN
   C := C.SUC ;
ELSE
   SG := C.SUC ;
   "Insert SG between G and FA" ;
   G := SG ;
END ;
```

PROOF 4

. C ≠ L within the loop A3 implies C.SUC ≠ NIL. Hence C.SUC.QUAL has a meaning.

. If G.QUAL ≠ C.SUC.QUAL, the assignment C := C.SUC maintains I(C). Otherwise the insertion of an element between G and FA also maintains I(C).

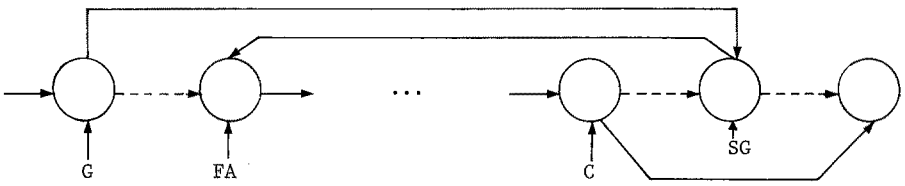
. I(FA) remains satisfied after the assignment G := SG since SG.QUAL = G.QUAL and since SG.SUC = FA will be true after execution of "Insert SG between G and FA".

. Let A designate an element such that A <= G and such that for each X satisfying A <= X <= SG we have A.QUAL = X.QUAL = G.QUAL. Since SG.QUAL = G.QUAL the condition (b) of I(G) is still satisfied after the assignment G := SG. In addition we have G<SG before as well as after the assignment. Hence the invariant I(G) is maintained.

. For each iteration of A3 either C progresses by one element or an element is removed from the sublist C-L. Hence the loop A3 terminates.

. This proof assumes that "Insert SG between G and FA" does not violate the nature of the list F-L and maintains I(C).

LEVEL 5 "Insert SG between G and FA"



The above diagram illustrates the transformation. Note that it is the first operation encountered so far which modifies the order of elements of the list F-L. Hence we must check that after the insertion, F-L is still a list satisfying the conditions P. In particular L must still designate the last element. Precautions are hence required in the case SG = L.

ALGORITHM A5

```
IF SG = L THEN
   L := C ;
END ;
C.SUC  := SG.SUC ;
SG.SUC := FA ;
G.SUC  := SG ;
```

PROOF 5

. If SG = L then the assignments L := C and C.SUC := SG.SUC imply that L.SUC = NIL

. A5 only modifies the successor fields of the elements C, SG, G and each of these elements is provided with a new successor. Hence each element has a successor in the resulting list F-G, SG, FA-C-L. It is clear also that no loop has been created.

. I(C) is maintained since the sublist FA-C is not modified and since C and G still designate the same elements as before.

FINAL ALGORITHM

```
(1)   G := F ;
(2)   UNTIL G = L LOOP
(3)     IF G.QUAL ≠ G.SUC.QUAL THEN
(4)       FA := G.SUC ;
(5)       C  := FA ;
(6)       UNTIL  C = L LOOP
(7)         IF G.QUAL ≠ C.SUC.QUAL THEN
(8)             C := C.SUC ;
(9)         ELSE
(10)            SG := C.SUC ;
(11)            IF SG = L THEN
(12)               L := C ;
(13)            END ;
(14)            C.SUC  := SG.SUC ;
(15)            SG.SUC := FA ;
(16)            G.SUC  := SG ;
(17)            G := SG ;
(18)        END ;
(19)      REPEAT ;
(20)    END ;
(21)    G := G.SUC ;
(22)  REPEAT ;
```

Insert SG between G and FA

Treat the successor of C

Remove synonyms of G which are successors of FA

Advance G while maintaining I(G)

3. CONCLUSIONS ON THE USE OF THIS METHODOLOGY IN AN EXPERIMENT

The example described above is part of a program of more than 2000 lines which deals with rather complex data structures. This program computes the displacements of the attributes of plex declarations in the LIS language. Given the descriptive facilities offered in LIS, this is a combinatorial problem analogous to a "knap-sack" problem and complicated by alignment requirements. The approach used is of heuristic nature. It delivers a "good" solution without any claim to optimality.

The program itself is not a "toy" program working in isolation. Much to the contrary it is integrated in a 25000 lines compiler. Its main interface with the latter is the dictionary of symbols, hence a complex data structure containing pointers. To summarize, this experiment dealt with a real program developed in a usual industrial context.

It is worth mentioning that when the work started we had no well defined idea of the algorithm to be used. The formulation of the solution and the program writing were both developed in parallel in a top-down structured approach. This means that several alternatives were initiated and then later rejected. As lower level refinements were produced, our intuition became more precise and we often knew enough to reject solutions which had appeared desirable at higher levels. The final 2000 lines are in fact what remains of more that the double of this number of lines written through several attempts.

In spite of the structured approach and of the use of informal proofs about twenty errors were found during program testing. The most frequent error could indeed have been avoided by a more careful approach. It was the omission of incrementations of cursors in loops. We had so much concentrated our attention on maintaining loop invariants that we had neglected to prove loop termination.

Two more serious errors were found corresponding to algorithm misconceptions. They required the rewriting of a hundred lines.

On the whole however, the error rate found in this programs is much smaller than for other parts of the compiler which we had judged to be simpler and for which this meticulous approach had not been used as systematically.

The whole work, designing, writing and testing, represented an effort of four man-months. In terms of productivity we must then conclude to the success of this practical experiment of structured programming with informal program proofs, expecially if we consider the complexity of the problem solved.

Although the construction of program proofs seems to proceed very slowly during the procedure by procedure writing of the program, it appears that the reduction in testing time is big enough to justify the approach on the whole.

REFERENCES

1. Dijkstra, E.W., "Goto statement considered harmful",
CACM, Vol. II (March 1968) pp 147-148.

2. Dijkstra, E.W., "Notes on structure programming",
in Structured Programming, pp 1-82, Academic Press, London (1972).

3. Floyd, R.W., "Assigning meanings to programs",
Proc. Am Math. Soc. Symp in Applied Math. 19 (1967) pp 19-31.

4. Hoare, C.A.R., "An axiomatic basis for computer programming",
CACM, Vol. 12 (October 1969) pp 576-583.

5. Hoare, C.A.R., "Proof of a program : FIND",
CACM, Vol. 14 (January 1971) pp 39-45.

6. Good, D.I., London, R.L., "Computer Interval Arithmetic : definition and proof of correct implementation",
JACM, Vol. 17,4 (October 1970) pp 603-612.

7. Wirth, N., "Systematic Programming : An Introduction",
Prentice Hall, Englewood Cliffs, N.J., (1973).

8. Manna, Z., "Mathematical Theory of Computation",
Mc Graw Hill, (1974).

9. Robinson, L., Levitt, K.N., "Proof techniques for hierarchically structured programs",
Stanford Research Institute, SRI technical report, (January 1975).

10. Infante, R., Montanari, U., "Proving Structured programs correct, level by level",
Proc. Int. Conf. on Reliable Software, (April 1975) pp 427-436.

11. Mills, H.D., "How to write correct programs and know it",
Proc. Int. Conf. on Reliable Software, (April 1975) pp 363-370.

12. Ichbiah, J.D., Rissen, J.P., Heliard, J.C., Cousot, P., "The system implementation language LIS",
CII technical report 4549 E/EN (December 1974).