UNSTRUCTURED SYSTEMATIC PROGRAMMING

by

M.H. van Emden

Research Report CS-76-09

Department of Computer Science

University of Waterloo
Waterloo, Ontario, Canada

February, 1976

UNSTRUCTURED SYSTEMATIC PROGRAMMING

by

M.H. van Emden *)

Abstract

Verification conditions (in the sense of Floyd's proof method) are shown to be identical with flowgraphs, a generalization of conventional flowcharts. The semantics of flowgraphs is defined, methods for proving partial and total correctness of flowgraphs are shown, and a translation of certain flowgraphs to FORTRAN is described.

The intended application of flowgraphs is as a systematic method for progressing from specifications to code (for example in FORTRAN). An example of this process is given, followed by a discussion of the implications for programming method, in particular for the choice of sequencing primitives.

Key Words and Phrases

programming method, structured programming, sequencing primitives, program correctness, assertions, verification conditions

CR categories 4.0, 5.24

*) Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

1. Introduction

Systematic programming may be taken to mean a method according to which the code of a program is obtained as the final stage of an orderly progression from an initially given specification of input-output behaviour. It is to Edsger W. Dijkstra that we owe to a large extent the knowledge that such a method is a worthwhile and feasible objective in programming.

To Dijkstra we also owe a method [2] designed to facilitate systematic programming. He emphasized that human intellectual limitations necessitate great care in the choice of primitives for sequence control. He concluded that in order to keep sequence control intellectually manageable it is wise to abstain from the use of goto statements and to rely instead on sequencing primitives which cause a program to become "well-structured".

For a proper understanding of the present paper it is important to distinguish between the objective of systematic programming and "structured programming", the method advocated by Dijkstra to achieve it. This particular method is only useful insofar as it is necessary to have intellectually manageable sequence control, and this is not always the case. For instance, when programming in a functional language like pure LISP, sequence control is of no concern to the programmer.

In this paper I describe the use of verification conditions (in the sense of Floyd's method of proof) as an aid to an orderly progression from specification to a program in "flowgraph" form. Verification conditions have in common with pure LISP that meaning is independent of any explicit sequence control. The advantage of certain sets of verification conditions is that they give rise to an equivalent FORTRAN program by a translation process which is in principle automatic and so simple that actual automation seems far from urgent. It is only in the translation process that sequence control becomes explicit, but at this point it has become so straightforward that to fear a problem one must take a very pessimistic view indeed of human intellectual limitations.

This use of verification conditions is an application of yet another method originating with Dijkstra [1]. Arguing that it is necessary to prove programs correct, he suggested that the greatest difficulties encountered in attempts at correctness proofs are caused by the particular approach where a program is completed first and a proof is attempted afterwards. Dijkstra showed that it is possible to develop a program and its proof in parallel; in this way the necessity to provide a proof does not need to be an additional burden on the programmer, but can actually facilitate the task of program construction. In his more recent work [3] Dijkstra carries the approach further, so that, in a sense, the proof comes before the program. In this paper I also follow this approach: the verification conditions of the proof are used as programs ("flowgraphs)".

In section 2 flowgraphs are introduced by a description of their form and meaning. Section 3 reviews Floyd's method of proof and shows the identity of flowgraphs and the verification conditions proving their partial correctness. I describe a method for proving termination which I believe to be useful in practice. Section 4 gives a worked miniature example of systematic programming with verification conditions. To convey the flavour of this as a problem-solving method, I wanted to give in some detail a "stream-of-thought" description of the development process. This takes up so much space that only a small example will be treated. A more realistic example of programming with verification conditions is the subject of a sequel and companion [4] to the present paper.

Section 5 gives the translation of flowgraphs to FORTRAN. Section 6 discusses the issue of program structure in greater detail. Sections 7 and 8 show that the generality of flowgraphs (admitting expression of "incomplete" and indeterminate algorithms) is useful because it enables the step-by-step approach to be followed in more directions than hitherto possible.

2. Flowgraphs

A *flowgraph* is a labelled directed graph having exactly one node (the *start node* S) without incoming arc and exactly one node (the *halt node* H) without outgoing arc. Each arc is labelled with a command. A *command* is a binary relation over a set of states or, equivalently, a set of pairs of states. The first element of such a pair is the *input state*; the second element is the *output state*. If c_1 and c_2 are commands their *product* $c_1;c_2$ is also a command: a pair (x,z) of states belongs to $c_1;c_2$ if and only if there exists a state y such that $(x,y) \in c_1$ and $(y,z) \in c_2$. The precise nature of the *states* need only be revealed in a definition of the commands of a flowgraph.

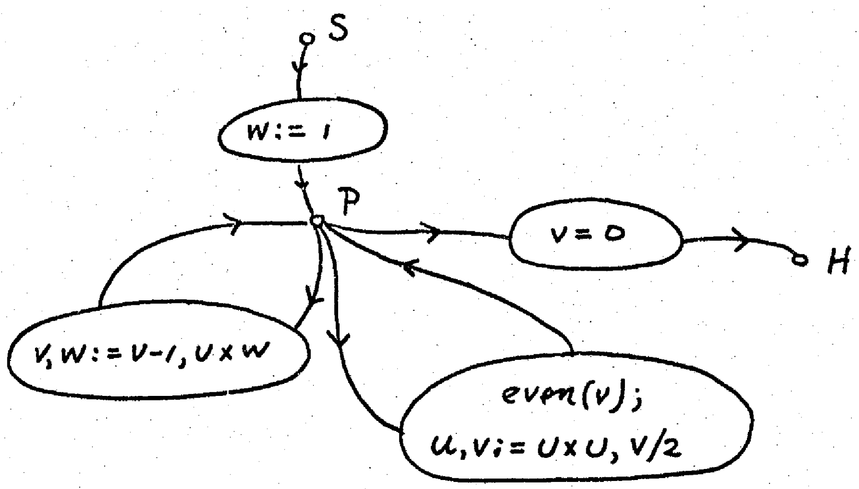
Example 2.1

In the set notation for the states and commands in this example, the variables u,v,w,u',v', and w' range over the set of rational numbers. The set of states is $\{(u,v,w)\}$

	mathematical notation for command	programming notation for command
a)	$\{((u,v,w),(u,v',w')) : v' = v-1 \ \& \ w' = u \times w\}$	v,w:= v-1,u x w
b)	$\{((u,v,w), (u',v',w)) : u' = u \times u \ \& \ v' = v/2\}$	u,v:= u x u, v/2
c)	$\{((u,0,w), (u,0,w))\}$	v = 0
d)	$\{((u,v,w), (u,v,w)) : v \neq 0\}$	v ≠ 0
e)	$\{((u,v,w), (u',v',w)) : \text{even } (v) \ \& \ u' = u \times u \ \& \ v' = v/2\}$	even (v); u,v:= u x u, v/2

Box 2.1 Some commands

The commands are of the kinds used by Dijkstra [3]: a) and b) are parallel assignations, c) and d) are guards, and e) is a guarded command. Only the guard may need some additional explanation. A guard is a command, hence a set of pairs of states. In particular, a guard is a command where the input state is the same as the output state. It contains only those pairs of equal states where the guard, regarded as a condition, is true. The use of such commands to achieve the effect of a test will become clear after the definition of a computation.



Flowgraph 2.1: An example illustrating the use of some of the commands in Box 2.1.

The purpose of a flowgraph is to characterize a set of computations. A *computation* is a sequence of (node, state)-pairs

$$(N_o, x_o), \ldots, (N_i, x_i), \ldots$$

where N_o is S(the start node)and where each pair after the first is a successor of the pair before it in the sequence: for $i \geq 1$, a pair (N_i, x_i) is a *successor* of (N_{i-1}, x_{i-1}) if and only if $(x_{i-1}, x_i) \in C$, the command labelling an arc from N_{i-1} to N_i.

The state x_o is called the *start state*. In a finite computation the last pair may be such that no successor for it exists. If the node in the last pair of a finite computation is the halt node H then such a computation is a *terminated computation*, otherwise it is a *blocked computation*.

See Box 2.2 for example computations of Flowgraph 2.1. All three computations shown have the start state (2,10,w), and yet they are different. A flowgraph, like this one, is called *indeterminate* whenever a (node,state)-pair can have two or more successors.

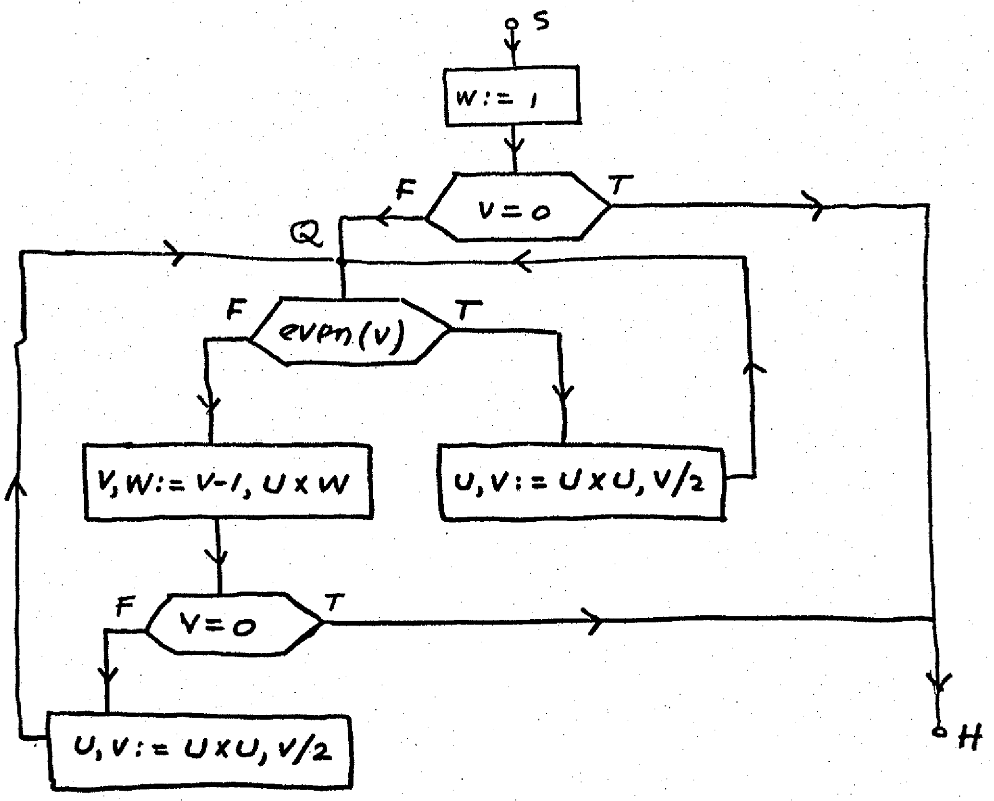
i	computation 1		computation 2		computation 3	
	N_i	x_i	N_i	x_i	N_i	x_i
0	S	(2,10,w)	S	(2,10,w)	S	(2,10,w)
1	P	(2,10,1)	P	(2,10,1)	P	(2,10,1)
2	P	(4,5,1)	P	(4,5,1)	P	(4,5,1)
3	P	(4,4,4)	P	(4,4,4)	P	(4,4,4)
4	P	(16,2,4)	P	(16,2,4)	P	(16,2,4)
5	P	(256,1,4)	P	(16,1,64)	P	(16,1,64)
6	P	(256,0,1024)	P	(16,0,1024)	P	(16,0,1024)
7	H	(256,0,1024)	H	(16,0,1024)	P	$(16,-1,2^{14})$
8					P	$(16,-2,2^{18})$
9					⋮	⋮

Box 2.2: Three computations of Flowgraph 2.1.

The terminated computations are useful for defining a flowgraph as a command: a pair (x,y) of states is in the command if and only if there exists a terminated computation with (S,x) as first and (H,y) as last (node, state) pair. Such a command may be said to represent the input-output behaviour of the flowgraph.

The definition of a computation makes clear how guards behave like tests in restricting the possibilities for a computation to proceed. If all commands were defined for all input states, all paths would be computations. The guards, like tests in conventional flowcharts, select some paths as computations.

In order to illustrate the relationship between guards and tests, Box 2.3 shows a conventional flowchart equivalent to Flowgraph 4.7.



Box 2.3: A conventional flowchart equivalent with Flowgraph 4.7.

3. Correctness of flowgraphs

The purpose of the method of Floyd [4] is to prove a property of the behaviour of a program written as a flow diagram. The method applies to flowgraphs as well, as will now be explained.

By an *assertion* I will mean a set of states. According to the method of Floyd there is associated with each node an assertion which I will denote by the same symbol as the associated node; it will be clear from the context which is meant. The assertions are said to *verify* the flowgraph if for each arc the *verification condition*

$$L_1;C \subseteq C;L_2$$

holds, where C is the command labelling the arc from node L_1 to node L_2.

The use of the product symbol ";" in this context needs some further explanation. Until now it was only defined as an operation on commands (binary relations) and here it is used as an operation on a command and an assertion. $L_1;C$ is a binary relation over states. A pair of states $(x,y) \in L_1;C$ if and only if $x \in L_1$ and $(x,y) \in C$. Similarly, $(x,y) \in C;L_2$ if andonly if $(x,y) \in C$ and $y \in L_2$. Thus, the verification condition above states the property of C that, whenever the input x to C is in L_1, an output y, if one exists, is in L_2.

Theorem: If, for a verified flowgraph, a (node, state) pair (L,x) occurs in a computation with start state $x_o \in S$, then $x \in L$, where this time L is the assertion associated with node L.

Proof: Let (L_{i-1},x_{i-1}), (L_i,x_i) be two successive pairs in the computation. By the definition of a computation, $(x_{i-1},x_i) \in C_i$, where C_i is the command labelling an arc from L_{i-1} to L_i in the flowgraph. By the supposition that the assertions verify the flowgraph, $L_{i-1};C_i \subseteq C_i;L_i$. Suppose now that $x_{i-1} \in L_{i-1}$; then $x_i \in L_i$. Apparently, if in a computation of a verified flowgraph $x \in L$, then the same holds for all subsequent pairs. It was assumed that $x_o \in S$.

Note that the theorem concludes something about all pairs in all computations with initial state in S, no matter whether a computation be terminated, blocked, or infinite. A common application of the theorem is restricted to just the initial and final pairs in the terminated computations only: a verified flowgraph has the property that whenever the initial state $x_o \in S$, the final state of a terminated computation (when such a one exists) must be in H. Such a property is usually called "partial correctness with respect to S and H

For a verification condition

$$L_1;C \subseteq C;L_2$$

it is convenient, especially when C is not very short, to use the following variant of a notation due to C.A.R. Hoare:

$$\{L_1\}\ C\ \{L_2\}$$

<u>Example 3.1</u>

The verification conditions for Flowgraph 2.1 are:

$$\begin{array}{ll} \{S\} & w := 1 \{P\} \\ \{P\} & v = 0 \ \{H\} \\ \{P\} & \text{even}(v);\ \ u,v := u \times u,\ v/2 \{P\} \\ \{P\} & v,\ w := v-1,\ u \times w \{P\} \end{array}$$

The following assertions verify Flowgraph 2.1:

$$\begin{aligned} S &= (u=u_o \ \&\ v=v_o) \\ P &= (w \times u^v = u_o^{v_o}) \\ H &= (w = u_o^{v_o}) \end{aligned}$$

Therefore, applying the theorem, it follows that partial correctness holds with respect to S and H, that is, in the final state of a terminated computation, with initial state in S, the value of the variable w is $u_o^{v_o}$.

<u>End of example 3.1</u>

Note that a verification condition specifies a labelled arc of a flowgraph: it exhibits the command labelling the arc and it specifies the initial and final node of the arc. Thus, any set of verification conditions uniquely determines a flowgraph (apart from possibly present isolated nodes, which can be omitted without affecting the set of computations). The definition of a computation determines the set of computations, hence the behaviour of a flowgraph. Thus a set of verification conditions is itself a program, which is by definition partially correct with respect to assertions satisfying them. As we shall see in the next section, such a program may need some improvement, which can be carried out in a systematic fashion within the formalism of flowgraphs or verification conditions.

The question whether a flowgraph has only terminated computations (with initial states in a given assertion), or perhaps also blocked or infinite ones, concerns an aspect of correctness which is quite different from the one discussed above. It is often possible to design a flowgraph in such a way that it is easy to prove that the set of computations, with initial state in a given assertion S, contains terminated computations only. I will discuss the situation where such a proof can be given in two parts: one part proves the absence of infinite computations and another proves the absence of blocked computations.

For a proof of the absence of infinite computations the following method is often useful. The main idea is to introduce a function of the state, named the "counter". If we can show that the counter can only assume nonnegative integral values, is never incremented, and is decremented sufficiently often, then the absence of infinite computations follows. "Sufficiently often" is made more precise by requiring that the counter be decremented whenever a path in a given "cut set" of paths is traversed in a computation; a set T of paths is a *cut set* if and only if each infinite path starting at the start node has infinitely many occurrences of paths in T.

The requirement that the counter is always a nonnegative integer must be proved. The verification conditions are useful here because their truth proves something about every state in every computation (starting from somewhere in S), no matter whether the computation is terminated, blocked, or infinite. In particular, if every assertion is included in the set of states where the counter is a nonnegative integer, and if such assertions verify the flowgraph, then we can conclude that for every state in every computation starting from S, the counter is a nonnegative integer.

To summarize, the absence of an infinite computation starting in S may be concluded from the following premisses:

1) each node is associated with an assertion included in the set of states where the counter is a nonnegative integer
2) these assertions verify the flowgraph
3) no arc is labelled with a command that increments the counter
4) there is a cut set T such that every path in it decrements the counter when executed; more precisely: counter(x) > counter(y) whenever the pair of states $(x,y) \in C_1;\ldots;C_n$, where $C_1,\ldots,C_n$ are the commands labelling the successive arcs of a path in T and where x is in the assertion labelling the initial node of the path.

For suppose on the contrary that an infinite computation

$$(S,x_o),\ \ldots,\ (N_i,x_i),\ \ldots$$

would exist and suppose $x_o \in S$. Because of 1) and 2) $x_i \in N_i$ and counter(x_i) is a nonnegative integer for $i = 0,1,\ \ldots$. Because of 3) the sequence of counter(x_i) is monotone nonincreasing. Because of 4) counter(x_{i+1}) < counter(x_i) for infinitely many i, which contradicts the fact that counter(x_i) is a nonnegative integer for all i.

Obviously, a verified flowgraph cannot have a blocked computation with a node N in its final pair if for every $x \in N$ there exists a y and a command C labelling an outgoing arc from N such that $(x,y) \in C$. Hence, if this fact holds for every node of a flowgraph except the halt node, we may conclude the absence of blocked computations starting from S.

Note that the condition can require termination of certain atomic commands. Take for example the common situation of a flowgraph having out of a node A only the two arcs with verification conditions

$$\{A\}\ g;\ C_1\ \{B_1\}$$
$$\{A\}\ \neg g;\ C_2\ \{B_2\}$$

where g is a guard and where commands C_1 and C_2 are not guards. The condition for the absence of blocked computations now requires that for all x, C_1 terminates if $x \in g\&A$ and C_2 terminates if $x \in (\neg g)\&A$.

Total correctness with respect to assertions ϕ and ψ will be taken to mean partial correctness with respect to ϕ and ψ in conjunction with the fact that no infinite or blocked computations exist with start state in ϕ. The use of a counter to show the absence of an infinite computation is due to Floyd [5] and Manna [8]. The only possibly new contribution here is the use of assertions to prove that the counter is a nonnegative integer. Dijkstra [3] has integrated the use of a counter with Floyd's method of assertions for proving partial correctness to obtain a method for proving total correctness.

4. Example of a systematic development of an algorithm

I have now explained enough about flowgraphs and their correctness proofs to demonstrate a simple example of the systematic development of an algorithm by means of flowgraphs. The method to be followed will not just write the correctness proof in parallel with the step-wise development of the program, but rather will try to complete the proof as far as possible before beginning to work on the program.

Starting from the specifications, I will collect assertions and verification conditions for commands which are expected to be useful in the as yet unknown program. When a satisfactory set of verification conditions has been found, there exists already a program in flowgraph form which is proved partially correct by the verification conditions. In this section I will show an example of development from specifications to a flowgraph which is totally correct with respect to these specifications. The next section discusses a correctness-preserving transformation from a class of flowgraphs to FORTRAN code.

It may be necessary to emphasize here that the development is not automatic, but systematic: the steps do not have to be necessary, but merely plausible. In fact, I cannot even make all steps plausible and I only claim that the development here is more systematic than in typical programming practice. The central assertion will be, as it were, conjured up like a white rabbit out of the magician's hat.

As the example I take the problem of raising a number u_o to the power of a nonnegative integer v_o. I assume that states are triples of values of variables called u,v, and w. I require that initially u and v have as their values the given numbers u_o and v_o (the input specification) and that on completion $w = u_o^{v_o}$ (the output specification). Taking into account the specifications only we can already exhibit a flowgraph which is partially correct with respect to them (see Flowgraph 4.1).

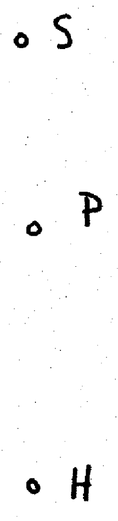
$$S = (u = u_o \& v = v_o)$$
$$H = (w = u_o^{v_o})$$

Flowgraph 4.1

Partial correctness holds vacuously for Flowgraph 4.1: it says something about terminated computations only, and this flowgraph does not have any. But the partial correctness, even in this case, is an important fact for the method I will follow, in which the final flowgraph will be the end of a sequence of partially correct flowgraphs, of which the empty flowgraph is the first and of which each successive flowgraph is an improvement over the previous one.

Flowgraph 4.1 needs improvement because it has no terminated computations. Inserting a nonempty command X such that {S} X {H}, may give an improvement, but I assume that no such command is available. Therefore at least one node, say P, with associated assertion will have to be introduced. In order to achieve at least one terminated computation, it is necessary that the assertion P should be such that at least nonempty commands X and Y can be found such that {S} X {P} and {P} Y {H}.

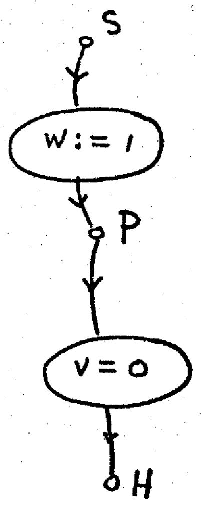
It will turn out that the invention of the right P is the major nonsystematic step in the development of the program. It contains, as it were, the "idea" of the ultimately obtained algorithm. I borrow this idea from Dijkstra's proof [2] of an exponentiation algorithm to get the verified Flowgraph 4.2.



$S = (u = u_o \ \& \ v = v_o)$
$P = (w \times u = u_o^{v_o})$
$H = (w = u_o^{v_o})$

Flowgraph 4.2

Note that P, although the "rabbit" of this example, is somewhat plausible: we will need a simple X and Y such that {S} X {P} and {P} Y {H}, which suggests that P should be in some sense a common generalization of S and H. For X and Y we find the commands shown in Flowgraph 4.3.



$S = (u = u_o \ \& \ v = v_o)$
$P = (w \times u^v = u_o^{v_o})$
$H = (w = u_o^{v_o})$

{S} w := 1 {P}
{P} v = 0 {H}

Flowgraph 4.3

The partially correct Flowgraph 4.3 has a terminated computation for some initial states (where v = 0) but not yet for any states where a nontrivial computation is required. This deficiency suggests the addition of at least one loop to the flowgraph with the purpose of decreasing a positive v to 0. Hence v should be tried as the "counter" of section 3; that is, the value of v in state x is counter(x). For a proof of the absence of infinite computations it is required that all assertions are included in the assertion which I shall call "nni (v)" and which is the set of states where v is a nonnegative integer. I change the assertions S,P, and H to achieve this and check that the verification conditions hold with the new assertions.

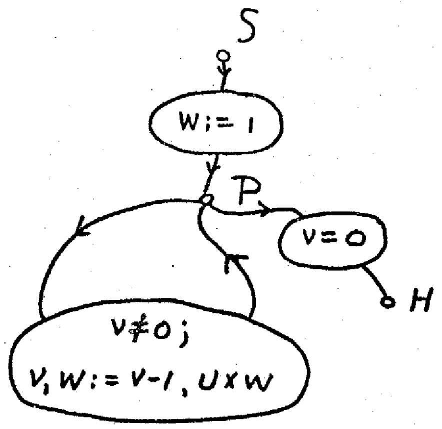
An arc from P to P must be labelled with a command X that satisfies {P} X {P}, in order to preserve partial correctness. For a simple proof of termination I will try to use a cut set of single-arc paths, which must then include as a path the arc labelled with X. So X must decrement the counter. These requirements suggest the use of the identity $w \times u^v = w \times u \times u^{v-1}$ to try to prove that $X = (v,w := v-1, u \times w)$ is a satisfactory command. However P implies that v is a nonnegative integer; this must also hold after the command. The straightforward way of making sure of this is to put X = (v,w:= v-1, u x w; $v \geq 0$). But commands with a guard following an assignation will give trouble in translating to FORTRAN or Algol. So it is better to use the equivalent (that is, being the same set of pairs of states) command

$$X = (v > 0;\ v,w := v-1,\ u \times w)$$

Because P implies nni(v) we might as well take for X

$$v \neq 0;\ v,w := v-1,\ u \times w$$

See Flowgraph 4.4. This way of arriving at a guarded command with the purpose of ensuring termination may be regarded as an application of Dijkstra's method for the design of "property terminating constructs" [3].



$$S = (u = u_o \ \&\ v = v_o \ \&\ nni(v))$$
$$P = (w \times u^v = u_o^{v_o} \ \&\ nni(v))$$
$$H = (w = u_o^{v_o})$$

$$\{S\}\ w := 1\ \{P\}$$
$$\{P\}\ v = 0\ \{H\}$$
$$\{P\}\ v \neq 0;\ v,w := v-1,\ u \times w\ \{P\}$$

Flowgraph 4.4

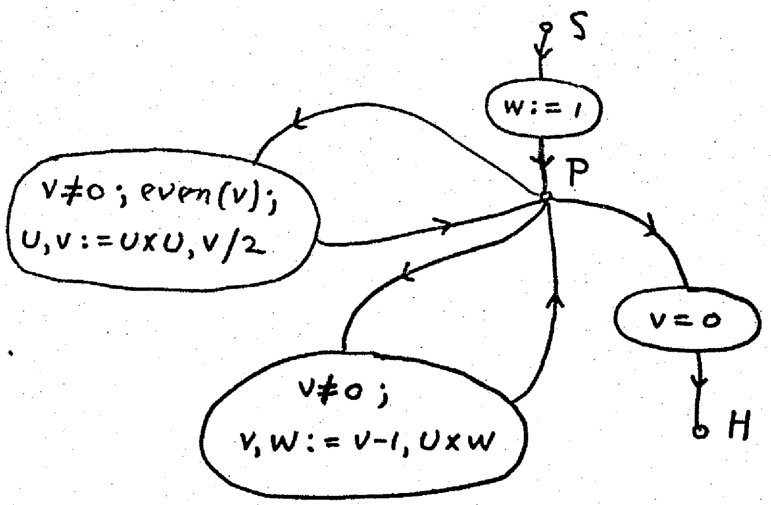
It is easily ascertained that Flowgraph 4.4 is partially correct with respect to S and H, and that it has no blocked or infinite computations starting from S. But the efficiency leaves a lot to be desired. Fortunately the method of flowgraphs allows improvements to be easily made. One is found by using another identity useful for decreasing v: $u^v = (u \times u)^{v/2}$. This suggests that another arc from P to P be introduced, labelled by the command u,v:= u x u, v/2. The requirement that v remain integral is taken into account by refining the command to u,v:= u x u, v/2; integer (v)

The desirability of having guards before assignations is the reason for using instead the equivalent command

even(v); u,v:= u x u, v/2

The requirement that the new arc be included in a cut set, as a path of length 1 and that v must therefore be decremented by the command labelling it, forbids that v = 0. Hence the new arc must be labelled with

$v \neq 0$; even(v); u,v:= u x u, v/2



$S = (u = u_o \ \& \ v = v_o \ \& \ \text{nni} \ (v))$
$P = (w \times u^v = u_o^{v_o} \ \& \ \text{nni} \ (v))$
$H = (w = u_o^{v_o})$

$\{S\}$ w:= 1 $\{P\}$
$\{P\}$ v = 0 $\{H\}$
$\{P\}$ $v \neq 0$; v,w:= v-1, u x w $\{P\}$
$\{P\}$ $v \neq 0$; even(v); u,v:= u x u,v/2 $\{P\}$

Flowgraph 4.5

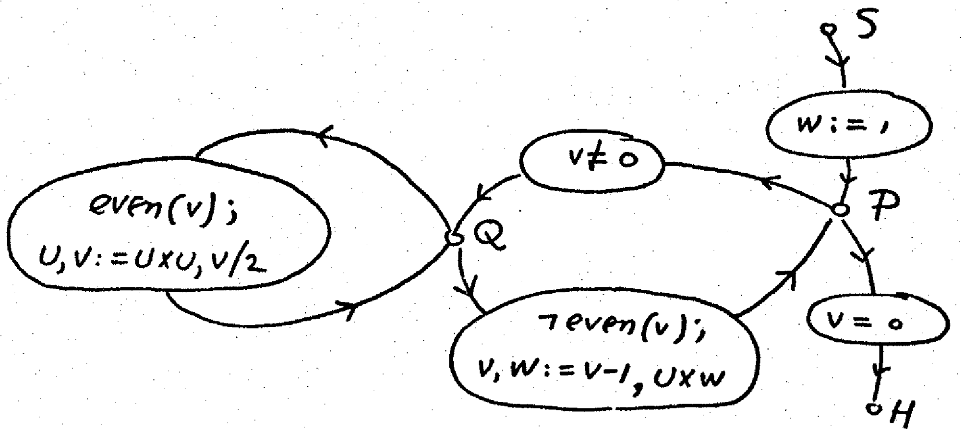
Note that the resulting Flowgraph 4.5 is indeterminate: efficient computations have been added to the set of computations of Flowgraph 4.4, but they have not replaced them. In order to exclude the inefficient computations it must be enforced that the usually more effective reduction in the counter is performed whenever possible. To achieve this the guard ¬even(v) is inserted into the command

$$v \neq 0;\ v,w := v-1,\ u \times w$$

to give

$$v \neq 0;\ \neg\,\mathrm{even}(v);\ v,w := v-1,\ u \times w$$

Two minor flaws remain: one is the fact that we have two commands beginning with the same guard $v \neq 0$ and starting from the same node P. It is easily seen how the improved, equivalent Flowgraph 4.6 is obtained from Flowgraph 4.5.



$S = (u = u_o \ \&\ v = v_o \ \&\ \mathrm{nni}(v))$

$P = (w \times u^v = u_o^{v_o} \ \&\ \mathrm{nni}(v))$

$Q = (P \ \&\ v \neq 0)$

$H = (w = u_o^{v_o})$

$\{S\}\ w := 1\ \{P\}$

$\{P\}\ v = 0\ \{H\}$

$\{P\}\ v \neq 0\ \{Q\}$

$\{Q\}\ \neg\,\mathrm{even}(v);\ v,w := v-1,\ u \times w\ \{P\}$

$\{Q\}\ \mathrm{even}(v);\ u,v := u \times u,\ v/2\ \{Q\}$

Flowgraph 4.6

The other flaw, which persists in Flowgraph 4.6, is that whenever in a computation

$$(Q,x_i),\ (P,x_{i+1}),\ (Q,x_{i+2}),\ (Q,x_{i+3})$$

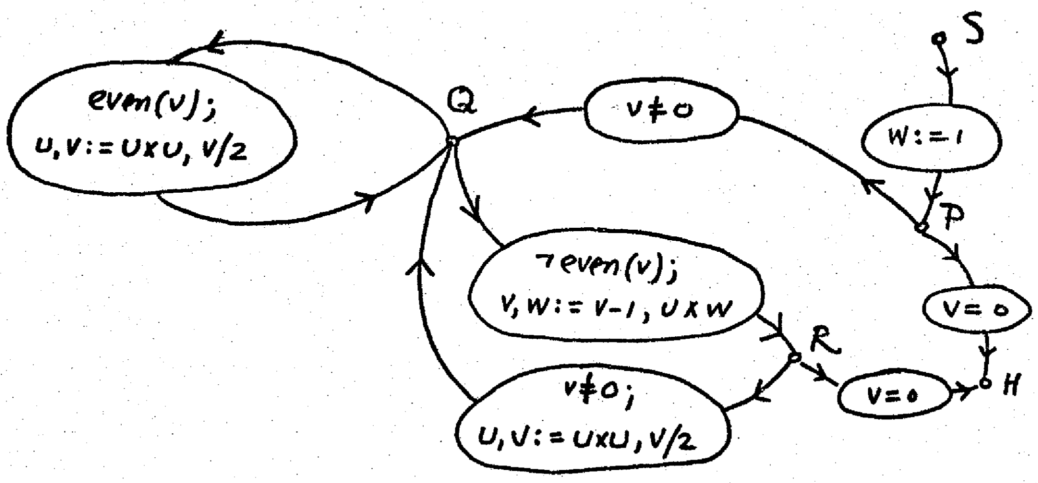
are successive (node, state) pairs, v is even in state x_{i+1} and therefore v is also even in $x_{i+2} = x_{i+1}$. But then the guard even(v) is superfluous. This situation is caused by the fact that

$$\{Q\}\ \neg\,\mathrm{even}(v);\ v,w := v-1,\ u \times w\ \{P\}$$

can be replaced by the stronger verification condition

$$\{Q\}\ \ \mathrm{even}(v);\ v,w := v-1,\ u \times w\ \{R\}$$

where $R = P \& even(v)$. Because we have $\{P\}\ v = 0\ \{H\}$, we certainly have $\{R\}\ v = 0\ \{H\}$. Because we have $\{P\}\ v \neq 0\ \{Q\}$ we certainly have $\{R\}\ v \neq 0\ \{Q\}$, but we even have $\{R\}\ v \neq 0;\ u,v{:=}\ u \times u,\ v/2\ \{Q\}$. It is only by including this last verification condition that we avoid the superfluous test. These changes give Flowgraph 4.7 as the end result of the example in systematic program development.



$S = (u = u_o \ \&\ v = v_o \ \&\ nni(v))$

$P = (w \times u^v = u_o^{v_o} \ \&\ nni(v))$

$Q = (w \times u^v = u_o^{v_o} \ \&\ nni(v) \ \&\ v \neq 0)$

$R = (w \times u^v = u_o^{v_o} \ \&\ nni(v) \ \&\ even(v))$

$H = (w = u_o^{v_o})$

$\{S\}\ w{:=}\ 1\ \{P\}$

$\{P\}\ v = 0\ \{H\}$

$\{P\}\ v \neq 0\ \{Q\}$

$\{Q\}\ even(v);\ u,v{:=}\ u \times u,\ v/2\ \{Q\}$

$\{Q\}\ \neg even(v);\ v,w{:=}\ v-1,\ u \times w\ \{R\}$

$\{R\}\ v \neq 0;\ u,v{:=}\ u \times u,\ v/2\ \{Q\}$

$\{R\}\ v = 0\ \{H\}$

Flowgraph 4.7

5. From Flowgraph to FORTRAN program

In order to be useful it is not necessary for a language to be implemented. There is a widely-known principle of good programming technique, which has been succinctly expressed by Kernighan and Plauger [6] in one of their 62 maxims:

> "Write first in an easy-to-understand pseudo language;
> then translate to whatever code you have to use."

Here flowgraphs play the rôle of "easy-to-understand pseudo language". Let us now assume, by way of example, that FORTRAN*) is the code we have to use.

Flowgraphs in many respects generalize conventional program languages. For instance, a flowgraph may be indeterminate, it may give rise to blocked computations, and it may contain commands of the form a;g, where a is an assignation and where g is a guard. It is vain to look for a useful translation to FORTRAN of flowgraphs with any such features; for the translation process to be described below I suppose they are absent, as in Flowgraph 4.7.

Both the description of the translation process and of its meaning will be rather sketchy. Just as flowgraphs, as described here, cannot be more than fragments of programs because no provision has been made for data declarations, so also will the translations in FORTRAN be fragments excluding declarations. I assume that there is a generally agreed semantics for FORTRAN according to which such a fragment corresponds to an input-output relation between states on entry to the fragment and states on exit from the fragment. I further assume that the data types and operators used in expressions in a flowgraph are defined with the same meaning in FORTRAN. The FORTRAN translation to be described below is intended to have the same input-output behaviour as the original flowgraph, given these two assumptions.

---

*) in this paper FORTRAN will mean ANSI standard FORTRAN.

Let the flowgraph to be translated be given as a sequence of verification conditions. Because their order is irrelevant for the flowgraph, we may assume the contiguity of all verification conditions with the same initial assertion. Let us call such a subsequence a *segment* and let us assume that the sequence of verification conditions begins with the segment of those having S as initial assertion.

The translation of the sequence of segments is the sequence of FORTRAN translations of the segments, followed by

```
h CONTINUE
```

In the description of the translation process, I use lower-case letters for items still to be translated to FORTRAN.

A segment

$$\{P\}\ C_1\ \{Q_1\}$$
$$\dots$$
$$\{P\}\ C_k\ \{Q_k\}$$

is translated to the FORTRAN text

$$p_1 \quad s_1$$
$$\dots$$
$$p_k \quad s_k$$
$$p_{k+1} \quad \text{CONTINUE}$$

Here the p_i are labels, distinct from each other and from those obtained by translating other segments.

Now

$$p_i \quad s_i$$

itself stands for a sequence of FORTRAN statements. If the i-th verification condition in the segment is

$$\{P\}\ g\ \{Q_i\} \text{ with } g \text{ a guard}$$

then

$$p_i \quad s_i \quad \text{is} \quad p_i \text{IF (g) GOTO } q_{i1}$$

where q_{i1} is the first label translating an occurrence of Q_i as initial assertion. If the i-th verification condition in the segment is

$$\{P\}\ a\ \{Q_i\} \text{ with } a \text{ an assignation,}$$

then

```
        p_i  s_i   is   p_i  a_1
                             ...
                             a_n
                             GOTO q_i1
```

where $a_I, \ldots, a_n$ are FORTRAN assignations (each to one variable at a time) jointly equivalent to a. If the i-th verification condition is

$$\{P\}\ g;a\ \{Q_i\} \text{ with } g \text{ a guard and } a \text{ an assignation,}$$

then

```
        p_i  s_i   is   p_i IF (.NOT.g) GOTO p_i+1
                        a_1
                        ...
                        a_n
                        GOTO q_i1
```

<u>Example 5.1</u> The verification conditions of Flowgraph 4.7 translate to:

```
        s_1  W = 1
             GOTO p_1
        s_2  CONTINUE
        p_1  IF (v = 0) GOTO h
        p_2  IF ( v ≠ 0) GOTO q_1
        p_3  CONTINUE
```

```
$\underline{q_1}$  IF (.NOT. $\underline{even(v)}$) GOTO $\underline{q_2}$
     U = U * U
     V = V/2
     GOTO $\underline{q_1}$
$\underline{q_2}$  IF (.NOT. $\underline{\neg even(v)}$) GOTO $\underline{q_3}$
     V = V - 1
     W = U * W
     GOTO $\underline{r_1}$
$\underline{q_3}$  CONTINUE
$\underline{r_1}$  IF (.NOT.  $\underline{v \neq 0}$) GOTO $\underline{r_2}$
     U = U * U
     V = V/2
     GOTO $\underline{q_1}$
$\underline{r_2}$  IF ($\underline{v = 0}$) GOTO $\underline{h}$
$\underline{r_3}$  CONTINUE
$\underline{h}$   CONTINUE
```

The underlined expressions still have to be translated. After doing this, and performing some obvious optimization, I obtain:

```
C  ASSERTION S
   10      W = 1
C  ASSERTION P
           IF (V.EQ.0) GOTO 40
C  ASSERTION  Q
   20      IF (MOD(V,2).NE.0) GOTO 30
           U = U * U
           V = V/2
           GOTO  20
   30      V = V - 1
           W = U * W
C  ASSERTION  R
           IF (V.EQ.0) GOTO 40
           U = U * U
           V = V/2
           GOTO 20
C  ASSERTION  H
   40      CONTINUE
```

The convention regarding the comments in the above program is that the assertion mentioned holds true just before executing the statement on the next line.

End of example 5.1

It seems to me that it is too much to ask for programs which are both understandable ("self-documenting") and efficiently executable in some generally available language. It also seems quite unnecessary to do so. By itself, the optimized FORTRAN program is probably hopelessly cryptic, but then it is not intended for human readers. Its documentation is the set of verification conditions of Flowgraph 4.7 together with the unoptimized FORTRAN program. Such documentation provides not only understanding, but even a proof of correctness.

In some situations the translation process as described here is unnecessarily cumbersome. Practical experience readily suggests shortcuts of which the value may be a matter of taste. For instance, in other examples I frequently encountered verification conditions like

$\{P_1\}\ g;a\ \{P_3\}$

$\{P_1\}\ \neg g\ \{P_2\}$

$\{P_2\}\ \ldots$

which I would change to

$\{P_1\}\ g'\ \{P_2\}$

$\{P_1\}\ \neg g';a\ \{P_3\}$

$\{P_2\}\ \ldots$

According to the rules given above, these translate to

$\underline{P_{11}}$ IF ($\underline{g'}$) GOTO $\underline{P_{21}}$

$\underline{P_{12}}$ IF (.NOT. $\underline{\neg g'}$) GOTO $\underline{P_{13}}$

$\underline{a_1}$

...

$\underline{a_n}$

GOTO $\underline{P_{31}}$

$\underline{P_{13}}$ CONTINUE

$\underline{P_{21}}$ ...

but I would remember to write immediately

```
    P11  IF (g') GOTO P21
         a1
         ...
         an
         GOTO P31
    P21  ...
```

6. Goto statements not necessarily harmful

Verification conditions have an easy translation to Algol 60. Let the verification conditions again be ordered in such a way that those with the same initial assertion are contiguous (and call the resulting subsequence again a segment). The translation of a set of verification conditions is a sequence starting with the translation of the segment with S as initial assertion, followed by the translations of any other segments, and ending with a dummy statement labelled H, the label translating the assertion H.

A segment of verification conditions

$$\{P\}\ C_1\ \{Q_1\}$$
$$\dots$$
$$\{P\}\ C_k\ \{Q_k\}$$

translates to

```
P: σ1;
   ...
   σk;
   goto P;
```

The presence of the final goto statement ensures that flowgraphs with blocked computations can also be translated. The translation is correct only for terminated and for infinite computations; a blocked computation of the flowgraph becomes an infinite computation in the Algol program.

If C_i is a guard γ then σ_i is
<u>if</u> γ <u>then</u> <u>goto</u> Q_i

If C_i is an assignation α then σ_i is
α;<u>goto</u> Q_i

If C_i is a guarded assignation γ ; α, then σ_i is
<u>if</u> γ <u>then</u> <u>begin</u> α;<u>goto</u> Q_i <u>end</u>

For example, the verification conditions for flowgraph 4.7 translate to the following fragment of an Algol-60*) program:

```
S:  w:=1; goto P;
    goto S;
P:  if  v = 0  then goto H;
    if  ¬v = 0  then goto Q;
    goto P;
Q:  if even(v)
    then begin  u:= u × u; v:= v÷2; goto  Q
         end;
    if ¬even(v)
    then begin  v:= v - 1; w:= u × w; goto  R
         end;
    goto  Q;
R:  if ¬(v = 0)
    then begin  u:= u × u; v:= v÷2; goto  Q
         end;
    if  v = 0 then goto H;
    goto  R;
H:
```

*) in actual fact "even" is not a standard function of Algol 60; I assume it is available as a library procedure

After some obvious optimizations the above program is reduced to:

```
S:  w:=1;
    if  v = 0  then goto H;
 Q: if even(v)
    then begin u:= u×u; v:= v÷2, goto  Q
         end;
    v:= v - 1; w:= u×w;
    if ¬(v = 0)
    then begin u:= u×u; v:= v÷2; goto  Q
         end;
 H:
```

Note that, apart from compound statements, the statement formats endorsed by structured programming [2,7] are of no use in the method described here. Goto statements are harmful only when the programmer has to explicitly manage the sequence control in his program. When relying on the goto statement for sequence control, it is notoriously difficult to avoid getting tangled up. Only then is structured programming useful because one has to make sequence control intellectually manageable.

In flowgraph programming it is not necessary to have sequencing intellectually manageable because there is no need for the programmer to manage it explicitly. In the verification conditions there is no sequencing. Only in the translation to a FORTRAN of Algol 60 program does sequence control appear: automatically, guaranteed correct, and, I have to add, almost entirely in the form of goto's. The flowgraph programmer is only involved in explicit sequence control for the trivial optimizations of the translated flowgraph.

The translation process guarantees that just before executing a goto a certain assertion holds; the goto then transfers to the label associated with that assertion: there is no harm in a goto if you know where you are going to.

7. Accretion as a principle of program construction complementary to refinement

For a problem too difficult to be solved in one step it is often useful to have a step-wise method that constructs successive approximations to the final solution. The "top-down" method described by Dijkstra [2], which was later to be called stepwise refinement by Wirth [9], is based on a process of abstraction where the solution to the original problem is conceived as a simple algorithm using, possibly very powerful, commands of a virtual machine. These commands are usually not implemented on the available machine: hence the implementation of each of them represents a programming problem in itself. In a successful application of the method, however, it is a self-contained problem of a considerably lower degree of complexity. This cycle represents the construction of one of the several layers in the ultimate, hierarchically structured, program, where the bottom layer is the one where the commands are finally implemented on the available machine.

The method of stepwise refinement attacks the problem of complexity by attempting a hierarchical decomposition of it: the total amount of complexity can be regarded as being somehow additively distributed over the layers, so that at each stage in the design process one only has to tackle a simple task, i.e. one has to handle only a small part of the total amount of complexity. These advantages have to be paid for both by the programmer and by the machine. The programmer has to design the interfaces between the virtual machines and the machine has to work the streams of information through the interfaces when executing the program. This means that one should try to keep the number of layers small.

The greater the chunk of complexity one can handle within a layer, the better. While recognizing the importance of stepwise refinement, I draw attention to the necessity of a complementary principle that helps to handle as much as possible of the complexity within a single layer, that is, without taking recourse to refinement. Flowgraphs are distinguished by the ease with which one can add nodes (with their assertions) and arcs (with their commands and required verification conditions) under preservation of partial correctness. The final flowgraph can be regarded as having developed by a process of *stepwise accretion,* which is the name I propose for the required complement to stepwise refinement. A flowgraph is a part of one layer in a hierarchically structured program. If all its commands are available on the actual machine, that layer is the bottom layer. If not, it is a separate task to program such a command as a flowgraph.

Accretion and refinement are complementary in the sense that one has to find a balance which avoids the introduction by refinement of too many layers but which also avoids unwieldy flowgraphs that have grown too large by carrying the accretion process too far. It is difficult to suggest an optimum size of flowgraph. The one derived in this paper is definitely below the optimum, whereas the flowgraph developed in the sequel [4] to this paper is probably much nearer.

8. The utility of stepwise accretion

In programming (and elsewhere) confusion results when one tries to do more than one thing at the same time, as happens, for instance, when one worries about efficiency before the design of the basic algorithm is completed. It helps to do as much as possible one thing at a time: "get it right before you make it faster", as Kernighan and Plauger [6] put it in another of their 62 maxims. And "getting it right" can itself be decomposed, with similar advantages.

There is much to be said for getting the program right even before one makes it determinate: see Flowgraph 4.5, which is correct, but not determinate. And I would even suggest to get the program right before making it do anything at all: the partially correct Flowgraph 4.1, 4.2, and 4.3 have no useful terminated computations, yet they are worth their place in the sequence of successive approximations.

The example treated in section 4 suggests the following design stages:

a) get the algorithm *complete*: add nodes or arcs **until** for each $x \in S$ the flowgraph has at least one terminated computation

b) get the algorithm *determinate* (if required): an indeterminate flowgraph can be made determinate by inserting suitable guards

c) get the algorithm *efficient*

All through these stages partial correctness is preserved: one must not only *get* it right *before* making it faster, but of course also *keep* it right *while* making it faster. Each of the above design stages may occur more than once, and not necessarily in the order given.

9. Acknowledgements

I have gratefully incorporated into this paper a large number of suggestions by J.D. Alanen of the University of Utrecht in the Netherlands. The work described in this paper is a spin-off from my research, supported (for another purpose) by the U.K. Science Research Council, which was done in the University of Edinburgh, initially in the Department of Machine Intelligence and later in the Department of Artificial Intelligence.

10. References

1. E.W. Dijkstra: Concern for correctness as a guiding principle for program composition.
In: The Fourth Generation, J.S.J. Hugo (ed.), **Infotech**, 1971.

2. E.W. Dijkstra: Notes on structured programming.
In: O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare: Structured Programming, Academic Press, 1972.

3. E.W. Dijkstra: A Discipline of Programming.
Prentice-Hall, 1976.

4. M.H. van Emden: A worked example in unstructured systematic programming.
Research report CS-76-27, Dept. of Computer Science, University of Waterloo.

5. R.W. Floyd: Assigning meanings to programs. Proc. Symp. Appl. Math. Vol. XIX, 19-32. J.T. Schwartz (ed.). Am. Math. Soc., Providence, 1967.

6. B.W. Kernighan and P.J. Plauger: The Elements of Programming Style.
McGraw Hill, 1974.

7. D.E. Knuth: Structured programming with goto statements.
Computing Surveys 6 (1974), 261-302.

8. Z. Manna: Termination of algorithms. Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon University, 1968.

9. N. Wirth: Program development by stepwise refinement.
Comm. ACM 14 (1971), 221-227.