Preface

In "Unstructured Systematic Programming" [4] I gave an exposition of programming with verification conditions without, however, giving more than a few cursory remarks comparing this method with others. As might have been expected, I did not get away with that. The present paper is an extensively revised version of "Unstructured Systematic Programming", containing in addition a more complete discussion of the comparisons mentioned above.

M.H. van Emden
Waterloo, 11 September 1977

ABSTRACT

This paper contains an exposition of the method of programming with verification conditions. Although this method has much in common with the one discussed by Dijkstra in "A Discipline of Programming", it is shown to have the advantage in simplicity and flexibility. The simplicity is the result of the method's being directly based on Floyd's inductive assertions. When programming with verification conditions one is able to express partially correct programs which fall short of total correctness by giving rise to "failed" finite computations. This possibility gives rise to the peculiar flexibility of the method which allows one to construct the required program as the final stage in a sequence of programs of which the first stage is directly obtained from the specification and in which each next stage is the result of a step towards total correctness while maintaining partial correctness with absence of infinite computations.

Index Terms: correctness-oriented programming, structured programming, verification, control structure, bottom-up programming, invariant assertions.

1. Introduction

The fact that programming is expensive and error-prone has been in the past decade a source of surprise, alarm, or even despair; it became the target of a large amount of activity now known as "software engineering". What is so special about programs that they give us that much trouble? Even if we can do no better than to say that it has to do with their complexity, then we can already distinguish two ways in which this phenomenon arises. In the first place there is the *amount* of complexity which large programs typically contain by virtue of sheer size. Then there is the possibility of *concentration* of complexity: even a very small program can be difficult to understand and to verify.

The first phenomenon, that of a large amount of complexity, may be tackled by the "top-down" method described by Dijkstra [2], which is based on a process of abstraction where the solution to the original problem is conceived as a simple algorithm using, possibly very powerful, actions of a virtual machine. These actions 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 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 actions are finally implemented on the available machine.

The top-down method 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 and that one should have within a single layer as great a chunk of the total amount of complexity as one can safely handle.

In other words, each action of a virtual machine should have as great an amount of complexity as one can safely handle without taking recourse to decomposition. To increase this amount we need a method that works in a direction opposite to the one of top-down programming, that is, we need "bottom-up" programming as well. The terminology of top-down versus bottom-up is not sufficiently informative, being based on the way we usually draw trees, which happens to be upside-down. What matters is that the top-down method *decomposes*, and that the complementary bottom-up method can be said to proceed by *aggregation*.

This paper contains an exposition of *programming with verification conditions* and a discussion of its methodological implications. The method allows decomposition in much the same way as other methods do; it is distinguished by the ease with which it allows step-wise aggregation under preservation of partial correctness. A discussion of other methodological aspects is to be found in sections 6 and 7 because we prefer to have an example to refer to. We review the required tools in sections 3 and 4, and give an example in section 5.

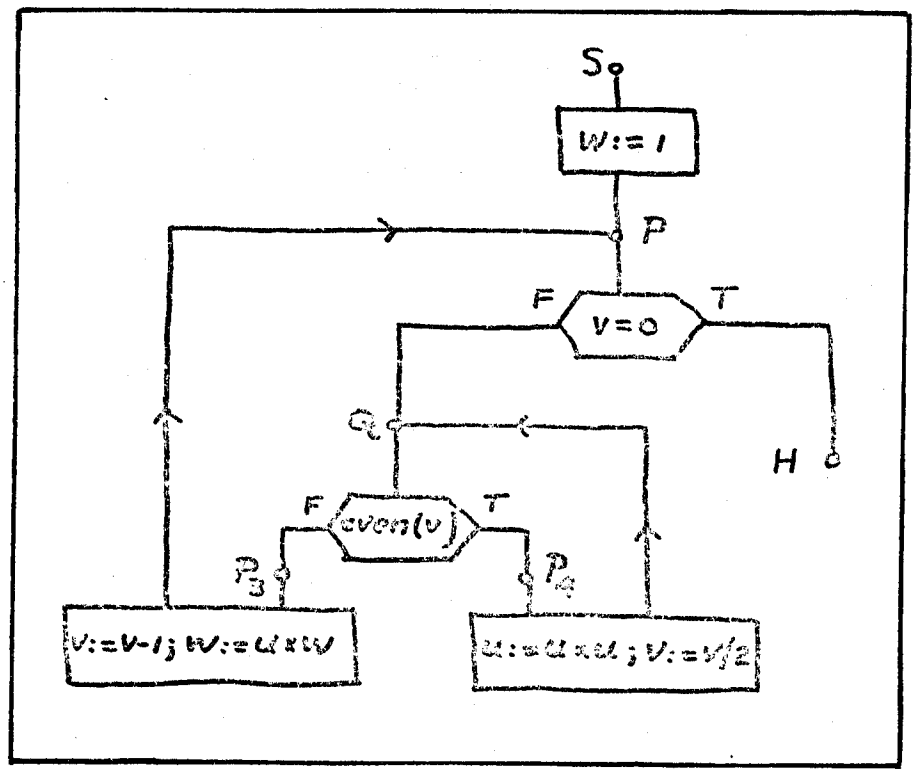
2. Related Work

Our method builds upon, simplifies, and extends Dijkstra's work [1,2,3]. Reynold's transition diagrams [13] are closely related to our flowgraphs. Hehner's paper [8] contains a critique of Dijkstra's do...od construct and some valuable hints on finding invariants. In [4] we gave a description and an example application of programming with verification conditions, but hardly any comparisons with other methods of systematic, correctness-oriented programming. In [5] we elaborated the relationship between programs and their verification conditions in the framework of first-order predicate logic. In [6] we characterized the meaning of both flowgraphs and verification conditions by means of minimal fixpoints.

Kowalski's paper [16] argues that it is possible and useful to decompose an algorithm into a "logic" component and a "control" component. One application of this decomposition is to explain the basic principle of our method: that a set of verification conditions is both a logical statement true independently of any sequence control, yet also a flowgraph program with a practically useful sequence control.

3. Flowcharts, Floyd's method, and flowgraphs

Consider the following example of a flowchart for an exponentiation algorithm.



Box 2.1

The "labels" S,P,Q,P_3,P_4,H are names of program points. A "state" is determined by the contents of the registers u,v, and w. Execution of the flowchart can be regarded as the construction of a computation, which is, for the purposes of this paper, the sequence of (label,state)-pairs recording which labels are successively encountered during execution and, for each label, the corresponding state. For example,

$(P_3,(8,5,1))$, $(P,(8,4,8))$, $(Q,(8,4,8))$, $(P_4,(8,4,8))$, $(Q,(64,2,8))$

is a subsequence of a computation of the flowchart.

According to Floyd's method [7,12] of proving correctness, an "assertion" is associated with each label, with the intention of ensuring that whenever execution reaches a label, the state satisfies the associated assertion. For example, Box 2.2 lists useful associations.

S	with	$u = u_o \ \& \ v = v_o$,
P	with	$w*u^v = u_o{}^{v_o}$,
Q	with	$P \ \& \ (v \neq 0)$,
P_3	with	$Q \ \& \ \neg \text{even}(v)$,
P_4	with	$Q \ \& \ \text{even}(v)$, and
H	with	$w = u_o{}^{v_o}$

Box 2.2

In Floyd's method, a flowchart is considered "verified" if execution reaching a label L with state s implies that the assertion associated with L is true of s. The proof that a flowchart is verified proceeds by induction. The basis of the induction is that the initial state satisfies the assertion associated with the start label, which can, in this example, always be satisfied by a suitable choice of u_o and v_o. The induction step consists of proving a number of "verification conditions", namely statements of the form

{ precondition} action { postcondition}

where the precondition and the postcondition are assertions. The meaning is that

> if state x satisfies "precondition" and
> "action" transforms x to y
> then state y satisfies "postcondition"

The verification conditions required for the flowchart in Box 2.1 are the following:

{ S}	w:=1 { P}
{ P}	successful execution of "v=0" { H}
{ P}	successful execution of "v≠0" { Q}
{ Q}	successful execution of "even(v)" { P_4}
{ Q}	successful execution of "¬even(v)" { P_3}
{ P_4}	u:=uxu; v:=v/2 { Q}
{ P_3}	v:=v-1; w:=uxw { P}

Box 2.3

where the assertions are as given in Box 2.2. The induction of Floyd's method proves that execution reaching H implies that $w=u_o^{v_o}$, that is, the result of the exponentiation algorithm is found in w. The induction has not proved that execution *will* reach H, which is indeed not always true; for example not when $v_o < 0$.

A result of the form

> if the initial state satisfies assertion S
> and the flowchart terminates
> then the final state satisfies assertion H

is called *partial* correctness because termination is assumed, not proved. Note that a verification condition is itself a statement of partial correctness for an action. A result of the form

> if the initial state satisfies assertion S
> then the flowchart terminates
> and the final state satisfies assertion H

is called *total* correctness.

According to Floyd's method termination is proved separately from partial correctness by showing that in each possible computation an integer, which is bounded from below, is decremented a sufficient number of times for an infinite computation to be impossible. This general principle can be used in several different ways. We shall return to termination later and explore first the connection between programs and their verification conditions.

From a mathematical point of view, an action can be a binary relation between input states and output states. Expressed in terms of sets, such

a relation R is the set of pairs (x,y) such that (x,y) in R iff y is a possible state after executing the action represented by R, when x is the state before. We will admit indeterminate commands, so y is *a possible* output state rather than *the* output state. Mathematically speaking, we may have that $(x,y_1) \in R$ and $(x,y_2) \in R$ and $y_1 \neq y_2$.

Relations can express another important computational phenomenon: it may be that for some x there exists no y such that (x,y) in R. This expresses the fact that for some input states the action of a command is "not defined". For example, if the input state x is such that w = 0, then for the action u:=u/w there is no corresponding output state y.

It will be useful to have a *dummy action* (so called after Algol 60's "dummy statement"): an action which does not change the state. These are modelled by the identity relation I: $(x,x) \in I$ for every state x.

If a_1 and a_2 are actions (represented by the relations R_1 and R_2), then $a_1;a_2$ is the action obtained by first executing a_1 and then a_2. The action $a_1;a_2$ is represented by the product of R_1 and R_2: (x,y) is in the product iff $(x,z) \in R_1$ and $(z,y) \in R_2$ for some state z.

Because every relation is a set and every set of pairs of states is a relation, every subset of a relation is a relation again. We are especially interested in relations which are subsets of the identity relation, such as

$$\{ (x,x) \mid B \text{ is true in state } x\}$$

where B is some assertion.

An action which is represented by such a subset of the identity is called a *guard*. If the input state to a guard satisfies the assertion, then the output state exists and is the same; otherwise no output state exists. If we write an assertion where one expects an action, then a guard is meant. For example

$$(v:=v/2);\ \text{integer}(v)$$

is an action and it has the same input-output behaviour as

$$\text{even}(v)\ ;\ (v:=v/2)$$

With these conventions we write the verification conditions of Box 2.3 as

$\{S\}$	$w:=1$	$\{P\}$
$\{P\}$	$v=0$	$\{H\}$
$\{P\}$	$v\neq 0$	$\{Q\}$
$\{Q\}$	$\text{even}(v);\ u:=uxu;\ v:=v/2$	$\{Q\}$
$\{Q\}$	$\neg\text{even}(v);\ v:=v-1;\ w:=uxw$	$\{P\}$

Box 2.4

These verification conditions are more succinct than the ones in Box 2.3 and they no longer have the same direct relationship to the flowchart in Box 2.1. We now define independently of flowcharts a representation of sets of verification conditions as labelled directed graphs, which we call *flowgraphs*. These turn out to be very similar to flowcharts.

For every assertion in a set of verification conditions there is a node in the corresponding flowgraph. For each verification condition there is an arc in the flowgraph directed from the precondition node to the postcondition node, labelled with the action of the verification condition. Let us also suppose that specifications for programs are given in the form of an input assertion and an output assertion. The input assertion is an assertion which the initial state may be assumed to satisfy. The output assertion is an assertion which the final state must satisfy. The node in the flowgraph corresponding to the input assertion (output assertion) is called the start node (halt node). In this paper we use the letters S and H, respectively. We assume that no verification condition has the input assertion as post-condition or the output assertion as precondition.

The flowgraph in Box 5.5 is the one corresponding to the verification conditions in Box 2.4.

Although flowgraphs are defined as a graphical representation of verification conditions, they also define sets of computations, hence, they are algorithms. The execution of a flowgraph can be pictured as a token tracing a path from the start node to the halt node through the graph, with the following constraint: the token carries a state and it may only pass through an arc if the action labelling the arc is defined for the state. As a result of passing through the arc the state on the token is changed as required by the action.

For a more precise definition of computation we first define the *successor relation* among (node,state)-pairs: (N',x') is a successor of (N,x) iff

there is an arc from N to N' and (x,x') is in the action labelling this arc. A finite computation is a finite sequence of (node,state)-pairs.

$$(N_o,x_o),\ldots,(N_k,x_k)$$

such that N_o is the start node

and (N_{i+1},x_{i+1}) is a successor of (N_i,x_i) for $i=0,\ldots,k-1$

and (N_k,x_k) has no successor.

If N_k is the halt node then the computation is *successful*, otherwise it is *failed*; x_o is the *start state* of the computation.

An infinite computation is an infinite sequence of (node,state)-pairs

$$(N_o,x_o)\ ,\ (N_1,x_1),\ldots$$

such that N_o is the start node

and (N_{i+1},x_{i+1}) is a successor of (N_i,x_i) for $i=0,1,\ldots$

Flowgraphs are more general than conventional programs in two respects. A (node,state)-pair may have more than one successor; in that case the flowgraph is *indeterminate*. A finite computation may be failed; because of this possibility *any set of verification conditions corresponds to some flowgraph*.

4. Correctness of flowgraphs

Any set of verification conditions corresponds to a flowgraph, and any flowgraph is a program in the sense that it defines a set of computations. Of course, the flowgraph may have, apart from successful computations, also failed or infinite ones; it may not even have any successful computations at all! This close correspondence between verification conditions and flowgraph programs is essential for the method described in this paper.

The following theorem shows that, if a flowgraph is viewed as a set of valid verification conditions, then it is this set that proves partial correctness according to Floyd's method.

Theorem: If, for a verified flowgraph, a (node,state)-pair (L,x) occurs in a computation with start state x_o satisfying the assertion associated with the start node, then x satisfies the assertion associated with L.

Proof: A flowgraph is verified when its verification conditions are valid. Let (N_i,x_i) and (N_{i+1},x_{i+1}) be successive pairs in a computation. By the definition of computation, $(x_i,x_{i+1}) \in C$, where C is the action labelling an arc from N_i to N_{i+1}. Let the corresponding verification condition be $\{P\}\ C\ \{Q\}$, where $P(Q)$ is the assertion associated with $N_i(N_{i+1})$. The validity of $\{P\}\ C\ \{Q\}$ implies that, if x_i satisfies P, then x_{i+1} satisfies Q □

Note that the theorem applies to computations of all kinds: successful, failed, infinite. A special case of the theorem for successful computations proves partial correctness with as precondition (postcondition) the assertion

associated with the start node (halt node). To prove in addition total correctness it is often useful to prove separately the absence of infinite computations and the absence of failed computations.

For a proof of the absence of infinite computations the following method (adapted from Floyd [7,12]) 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 "basis set" of paths is traversed in a computation; a set B of paths is a *basis set* if and only if each infinite path starting at the start node has infinitely many occurrences of paths in B.

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 successful, failed, 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 each state in any 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 basis set B 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 actions labelling the successive arcs of a path in B and where x is in the assertion labelling the initial node of the path. In other words, the paths of the basis set "cut all loops".

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) $x_i \in \mathbf{N}_i$ 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 failed computation with a node N in its final pair if for every $x \in N$ there exists a y and an action 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 failed computations starting from S.

5. Example I: Systematic construction of an exponentiation algorithm

The method of stepwise aggregation may be summarized as follows. Given as initial approximation to the required program only the specification in the form of a precondition and a postcondition, we successively add assertions and valid verification conditions, using the heuristic that failed computations be extended in the next approximation under the constraint of maintaining partial correctness and not introducing infinite computations. A statically ascertainable criterion, in this case the conditions for total correctness in the previous section, determines when the process of program construction by stepwise aggregation has been completed.

In this example the problem is to raise a number u_o to the power of a nonnegative integer v_o. We assume that states are triples of contents of registers called u,v, and w. The specification calls for a program which is totally correct with respect to precondition

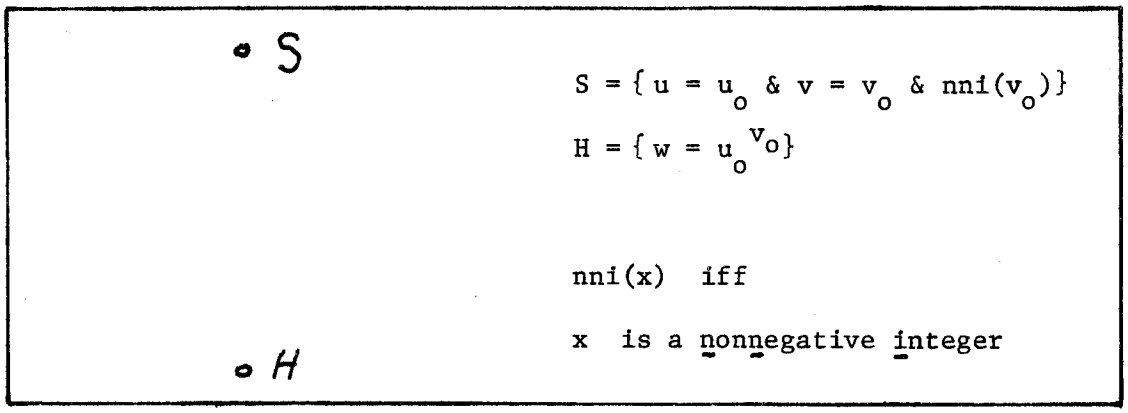
$$S = \{ u=u_o \ \& \ v=v_o \ \& \ nni(v_o) \},$$

where nni says whether its argument is a nonnegative integer, and postcondition

$$H = \{ w=u_o^{v_o} \}.$$

At this stage we have as approximation to the solution the following flowgraph, which is already partially correct. We maintain partial correctness with absence of infinite computations throughout the entire sequence of approximations,

and attempt to get closer to total correctness at each next step.



Box 5.1

The flowgraph in Box 5.1 needs improvement because it has no successful computations. Inserting a non-empty action X such that $\{S\}\ X\ \{H\}$ may give an improvement, especially when X is $w:=u_o{}^{v_o}$, but let us assume that no such action is possible. Therefore at least one other node, with associated assertion, will have to be introduced. In order to obtain at least one successful computation, it is necessary that the assertion P be such that nonempty commands X and Y can be found such that $\{S\}\ X\ \{P\}$ and $\{P\}\ Y\ \{H\}$.

As it turns out, the invention of the right P is almost all there is to invent in the algorithm ultimately obtained. If there would be a way of formally deriving P from S and H then it would be conceivable that the entire algorithm is constructed automatically. In the absence of such a formal derivation we assume P to be given as the assertion

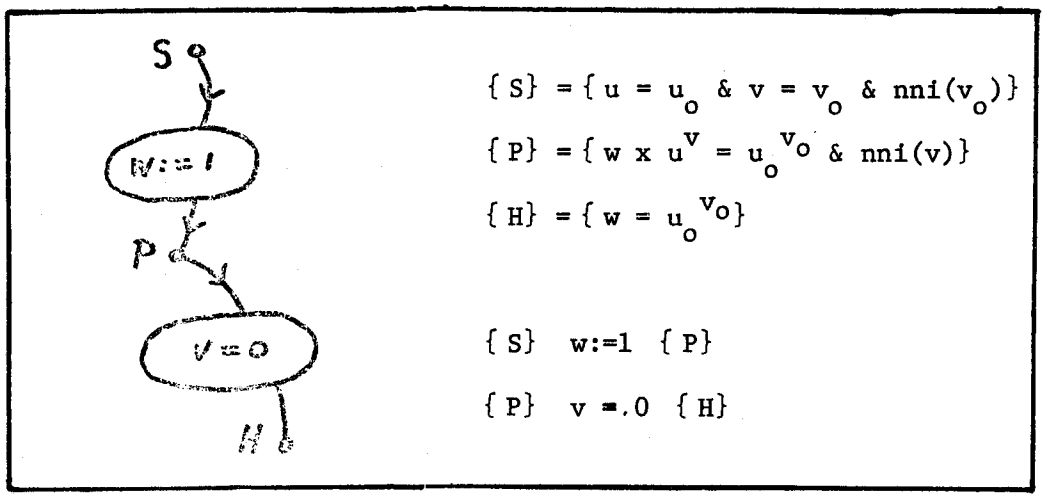
$$w \times u^v = u_o^{v_o}$$

as, for example, in Dijkstra's proof [3] of an exponentiation algorithm. We do not claim to have anything to contribute to solving the important problem of obtaining intermediate assertions. But even with this problem unsolved, reasoning in terms of assertions to find the required intermediate assertions is a much better method for constructing a correct program than directly manipulating program components, as seems to be the alternative. Therefore we will in this example bring some of the existing folklore [14,3,8] to bear upon the problem of at least making *plausible* the choice of P, so as hopefully to facilitate invention in future similar situations.

The fact that there is no useful X such that $\{S\}\ X\ \{H\}$ suggests that H is too hard to achieve directly. A commonly used heuristic is to *divide* (so as hopefully to *conquer*) into two easier goals, say H_1 and H_2. In easy cases, H_1 and H_2 are independent in the sense that H can be achieved by first achieving H_1 and then H_2. When H_1 and H_2 are not independent, achieving H_2 spoils the already obtained solution to H_1, and vice versa. But if we are lucky, and patient enough not to require H_2 to be achieved in a single step, we can take a *step towards* H_2 and then adjust the solution so that it still satisfies H_1. In that case we have the familiar iterative scheme where H_1 is an *invariant* and H_2 a stopping criterion.

The choice of P can be made somewhat plausible by regarding $\{H\} = \{w=u_o^{v_o}\}$ as a conjunction of two assertions $\{H_1\} = \{wxu^v = u_o^{v_o}\}$ and $\{H_2\} = \{v=0\}$.

H_1 can then be the intermediate assertion P which can be achieved initially by w:=1 and H_2 is achieved by passing the guard v=0. This gives not only as next approximation the flowgraph in Box 5.2, but it also suggests that v be used as counter for proving termination because a decrease in v is an obvious interpretation of "a step towards achieving H_2". Hence the conjunct nni(v) which henceforth appears in all nonhalt assertions.



Box 5.2

The flowgraph in Box 5.2 is again partially correct but only slightly less vacuous than the previous one: there are still not enough successful computations. We have not yet used the possibility of taking a step towards achieving H_2 under invariance of H_1. Let us first try a single command X satisfying {P} X {P} and such that no infinite computations are introduced. For a simple proof of termination we try a basis set of single-arc paths, which must then include as a path the arc labelled X. The identities

$$wxu^v = w \times (uxu^{v-1}) = (wxu) \times u^{v-1}$$

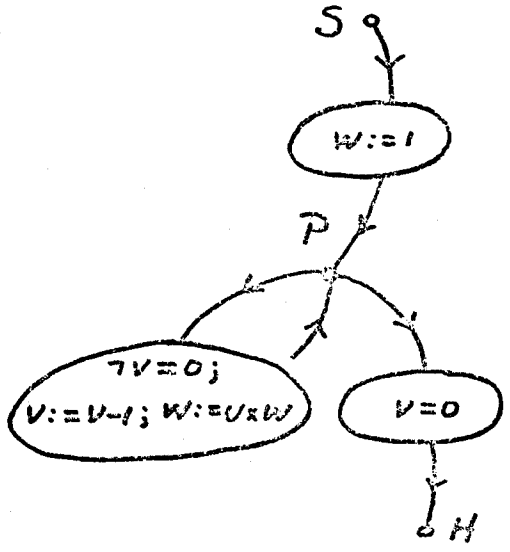
are useful for preserving P while decrementing v: they suggest trying for X the command v:=v-1; w:=wxu. However, in P v has to be nonnegative; this must also hold after X. The straightforward way of ensuring this is to put

$$X = (v:=v-1;\ w:=uxw;\ v \geq 0) \qquad \ldots(5.1)$$

But actions with a guard following an assignment will give trouble in translating to a conventional language. So it is better to use the equivalent

$$X = (v > 0;\ v:=v-1;\ w:=uxw) \qquad \ldots(5.2)$$

Because P implies nni(v) we might as well take $v \neq 0$ instead of $v > 0$, as in the flowgraph in Box 5.3.



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

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

$\{H\} = \{w = u_o{}^{v_o}\}$

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

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

$\{P\}$ v ≠ 0; v:=v-1; w:=uxw $\{P\}$

Box 5.3

The flowgraph in Box 5.3 is partially correct with respect to S and H, and has no failed or infinite computations sharting from S. Efficiency can be increased by using the identity

$$u^v = (uxu)^{v/2}$$

which usually decreases v faster. This suggests that another arc from P to P be introduced, labelled by the action

u:=uxu; v:=v/2

The requirement that v remain integral is taken into account by elaborating the action to

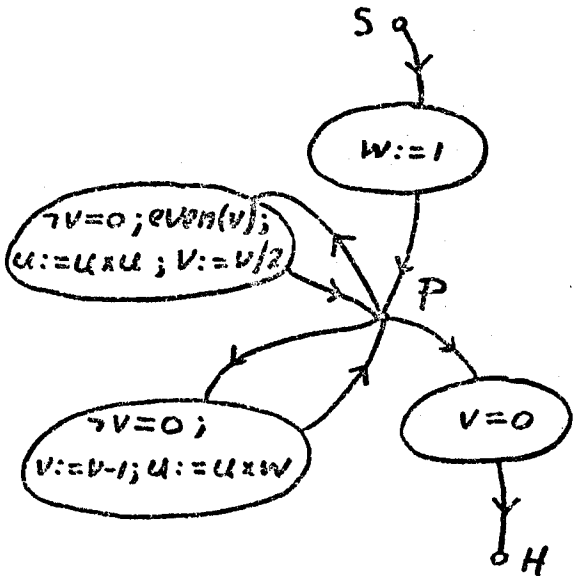
u:= u x u ; v:=v/2; integer (v) ...(5.3)

The desirability of having guards before assignments is the reason for using instead the equivalent action

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

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

v ≠ 0; even(v); u:=u x u; v:=v/2 ...(5.4)



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

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

$\{H\} = \{w = u_o{}^{v_o}\}$

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

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

$\{P\}$ $v \neq 0$; v:=v-1; v:= u x w $\{P\}$

$\{P\}$ $v \neq 0$; even(v); u:= u x u; v:=v/2 $\{P\}$

Box 5.4

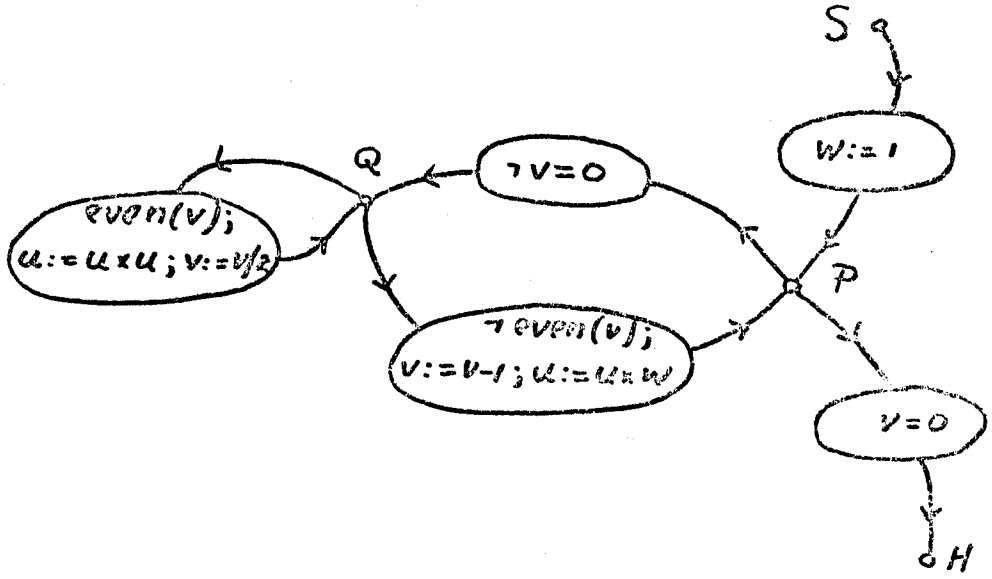
Note that the resulting flowgraph in Box 5.4 is indeterminate: efficient computations have been added to the set of computations of the previous flowgraph, but they have not replaced them. In order to exclude the inefficient computations it must be enforced that the usually more effective reduction in the counter is performed whenever possible. To achieve this the guard ¬even(v) is inserted into the action

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

to give

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

Two flaws remain: one is the fact that we have two actions beginning with the same guard $v \neq 0$ and starting from the same node P. An improved, equivalent flowgraph in Box 5.5 is obtained from the one in Box 5.4.



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

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

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

$\{H\} = \{w = u_o^{v_o}\}$

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

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

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

$\{Q\}$ ¬even(v); v:=v-1; w:= u x w $\{P\}$

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

Box 5.5

The other flaw, which persists in Box 5.5 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 activating the guard even(v) is superfluous. This situation is caused by the fact that

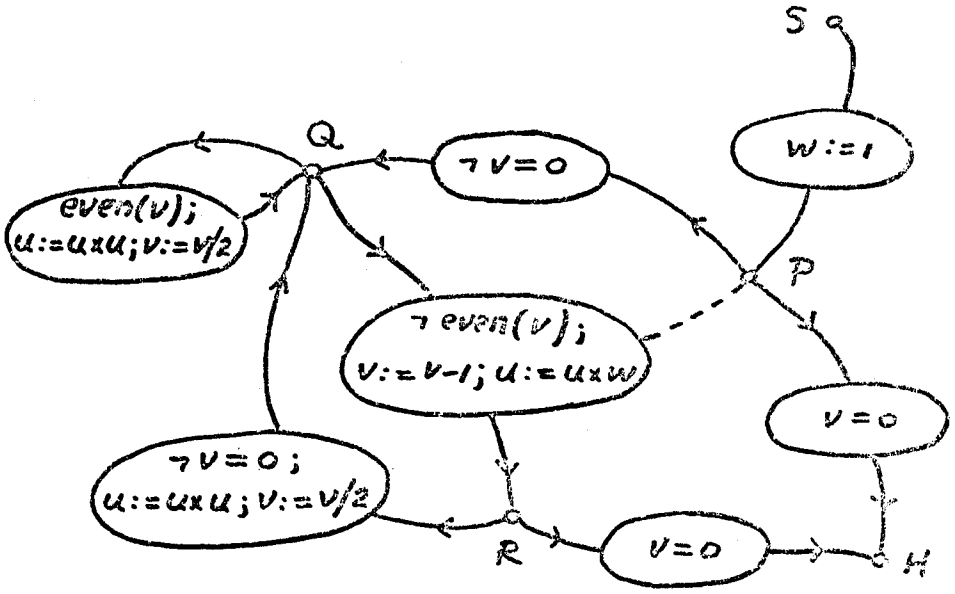
$$\{ Q \}\ \neg\text{even}(v);\ v := v-1;\ w := u \times w\ \{ P \}$$

can be replaced by the stronger verification condition

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

where $R = P \ \&\ \text{even}(v)$. 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 $\{ R \}\ v \neq 0;\ u := u \times u;\ v := v/2\ \{ Q \}$. It is only by including this last verification condition that we avoid the superfluous test. These changes give the flowgraph in Box 5.6 which is easily seen to be equivalent to the one in Box 5.7. Which of the two is preferred is largely a matter of taste.

All of the flowgraphs in Boxes 5.1,...,5.7 are partially correct and lack infinite computations. Those in Boxes 5.3, 5.5, 5.6, and 5.7 are determinate as well, lack failed computations, and their actions have a trivial translation to a conventional programming language. As an example, let us translate into Algol 60 the verification conditions of Box 5.7. The following translation rules are applicable for a useful class of verification conditions of which the flowgraphs must be without blocked computations.



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

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

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

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

$\{H\} = \{w = u_o{}^{v_o}\}$

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

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

$\{P\}$ v ≠ 0 $\{Q\}$

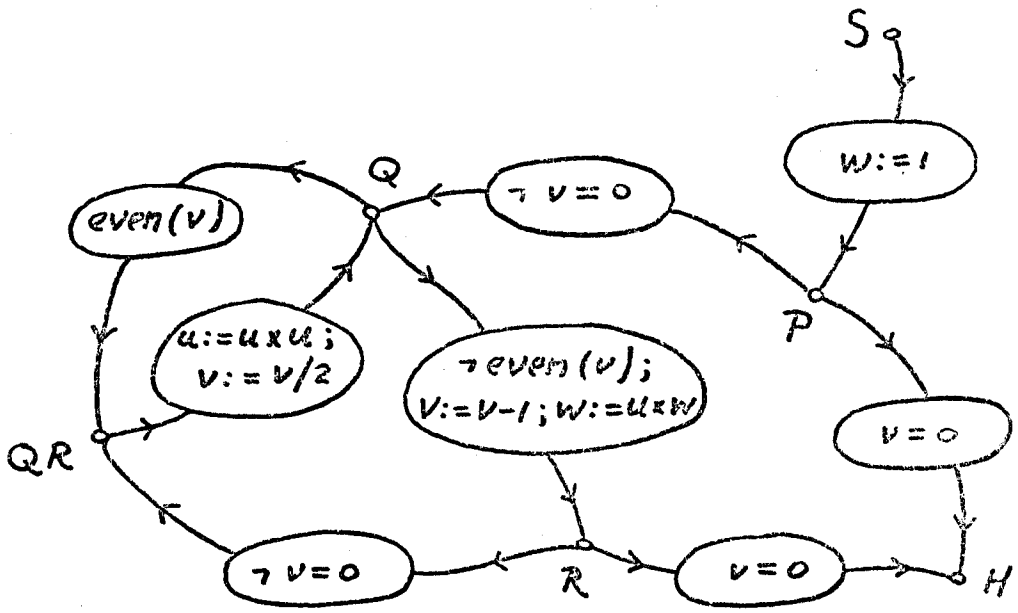
$\{Q\}$ even(v); u:= u x u; v:=v/2 $\{Q\}$

$\{Q\}$ ¬even(v); v:= v-1; w:= u x w $\{R\}$

$\{R\}$ v ≠ 0; u:= u x u; v:= v/2 $\{Q\}$

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

Box 5.6



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

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

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

$\{R\} = \{P \ \& \ even(v)\}$

$\{QR\} = \{Q \ \& \ R\}$

$\{H\} = \{w = u_o^{v_o}\}$

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

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

$\{P\}$ v ≠ 0 $\{Q\}$

$\{Q\}$ ¬even(v); v:= v-1; w:= u x w $\{R\}$

$\{Q\}$ even(v) $\{QR\}$

$\{QR\}$ u:= u x u; v/2 $\{Q\}$

$\{R\}$ v ≠ 0 $\{QR\}$

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

Box 5.7

Let the verification conditions be ordered in such a way that those with the same initial assertion are contiguous (and call the resulting subsequence a segment). Within a segment the order is insignificant. 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

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

translates to

$$P:\ \sigma_1;$$
$$\cdots$$
$$\sigma_k;$$

where the σ's are determined by the translation rules:

If C_i is a guard then σ_i is

$\underline{\text{if}}\ C_i\ \underline{\text{then goto}}\ Q_i$

If C_i is an assignment then σ_i is

C_i; $\underline{\text{goto}}\ Q_i$

If C_i is a guarded assignment γ; α, then σ_i is

$\underline{\text{if}}\ \gamma\ \underline{\text{then begin}}\ \alpha;\underline{\text{goto}}\ Q_i\ \underline{\text{end}}$

For example, the verification conditions for the flowgraph in Box 5.7 translate to the following fragment of an Algol-60 program:

```
S:  w:=1; goto P;
P:  if v=0 then goto H;
    if v≠0 then goto Q;
Q:  if ¬even(v) then begin v:=v-1; w:=uxw; goto R
                      end;
    if even(v) then goto QR;
QR:  u:=uxu; v:=v/2; goto Q;
R:  if v≠0 then goto QR;
    if v=0 then goto H;
H:
```

The application of some simple optimization rules gives:

```
S:  w:=1;
    if v=0 then goto H;
Q:  if ¬ even(v) then begin v:=v-1; w:=uxw
                        ;   if v=0 then goto H
                      end;
    u:=uxu; v:=v/2; goto Q;
H:
```

6. A methodological discussion of example I

Dijkstra [1] argued that it is necessary to prove programs correct and he suggested that the difficulties encountered in attempts at program verification are caused by the peculiar approach where a program is completed first and a proof is attempted afterwards. He showed that it is possible and advantageous to develop a program and its proof in parallel. He argued that 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 showed that an algorithm can be developed more easily by reasoning about assertions than by manipulating program components. Here the proof comes, in a sense, *before* the program.

The basic proof method relevant here is due to Floyd. Subsequently Hoare [9] used Floyd's method in a formal system for proving partial correctness with a rule of inference for each basic construct of the programming language. Dijkstra [3] expressed correctness in terms of "predicate transformers". Where Hoare applies rules of inference to obtain partial correctness, Dijkstra manipulates expressions denoting assertions and obtains total correctness. Each construct of the programming language is characterized by a predicate transformer which is a functional combination of the predicate transformers of the constituents of the construct.

In contrast with the above, our method of programming with verification conditions is directly based upon Floyd's original method [7,12], bypassing

any subsequent elaborations. This directness is made possible by the use flowgraphs, a form of program identical to a set of verification conditions. Results from our method are as provably correct as those from Hoare's or Dijkstra's, but we need no counterpart for Hoare's rules of inference of Dijkstra's calculus of predicate transformers for dealing with language constructs, simply because what little of language constructs we use, enters only in the translation phase from a flowgraph already proved totally correct. In particular, we claim that the problem-solving power of Dijkstra's use of the "wdec" predicate can be achieved in an easier way by informal reasoning of which the transitions from (5.1) to (5.2) and from (5.3) to (5.4) are examples.

We have argued above that programming with verification conditions is a *simplification* with respect to previous comparable methods. We now argue that our method has a *larger scope*: it allows the "divide and conquer" principle to be applied in more dimensions than previous methods do. Let us first review various ways in which the principle is useful in programming.

In programming (and elsewhere) confusion results when one tries to do more than one thing at a 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; yet being able to do so depends on a suitable *decomposition* of the task at hand. The first decomposition is suggested by Kernighan and Plauger's maxim [10]: "get it right before you make it faster". And the goal of getting it right can again be decomposed, with similar advantages.

There is much to be said for getting the program right before one makes it determinate: see the flowgraph in Box 5.4 which is correct, but not determinate. This decomposition, the second, we owe to Dijkstra [3]. And we suggest a third decomposition, namely to get the program right even before making it do anything at all: the partially correct flowgraphs in Boxes 5.1 and 5.2 have no useful computations. Flowgraphs can have failed computations; they may even fail to have any successful ones. This degree of freedom makes programming by stepwise aggregation possible: our initial program is the vacuously partially correct one directly obtained from the input-output specification; in each step an assertion or verification condition is added with the purpose of allowing failed computations to continue on toward success. The addition must maintain partial correctness and must not introduce any infinite computation.

Each of the above decompositions is useful. Programming with verification conditions uniquely contributes the third decomposition, while being equally suitable as other methods for the first and second decompositions.

Dijkstra's sequencing primitives [3] are unsuitable for the third decomposition; this is because of the, in our view unfortunate, way termination for do...od is defined. Note that if...fi can give rise to failed computations (Dijkstra refers to failure as "abortion"). The do...od construct cannot, by (Dijkstra's) definition, fail: the criterion for successful termination is implicitly, by default, determined by the guards, as the conjunction of their negations. We have argued elsewhere [6] that an explicit criterion for success, independent of the guards, is an improvement for goal-directed programming. Our proposal in [6] would make Dijkstra's primitives more suitable for the third decomposition.

7. Example II: A trade-off between complexity and efficiency of a program

The algorithm described by the flowgraph in Box 5.5 can be expressed in Dijkstra's language [3] as

```
w:=1;
do  v ≠ 0  →  do even(v)  →  u,v:=uxu,v/2
              od ;
              v,w:=v-1,uxw
od
```

It is significant that this algorithm is chosen rather than the more efficient one described by the flowgraph in Box 5.7. The latter is more complex in the sense of having more program points, which provide a sufficient "memory" for results of tests, so that no test needs to be duplicated.

In this section we will discuss the trade-off between having few program points (hence a simple algorithm which looks elegant in a "structured" language) and avoiding superfluous tests, which requires a certain minimum number of program points, in our method nodes of the flowgraphs. The fact that such a trade-off exists is typically swept under the rug in discussions promoting "structured" programming, where there seems to be a bias towards elegant algorithms performing superfluous tests.

We are aware that it is not usually worth the complexities of eliminating superfluous tests because anyway most programs are used only a few times (if at all). But in those rare cases where optimal efficiency *is* important, a systematic method must also help in discovering a satisfactory algorithm.

And the most important advantage of programming with verification conditions is that it allows one to cover fluently the entire spectrum between an efficient algorithm without duplicated tests and a simple algorithm of which the flowgraph has few nodes. The example discussed in this section is chosen because the spectrum is rather wide.

Let us consider the problem of merging two sorted input files of numbers (a left file called lft and a right file called rht) into a single, sorted output file called tpt. We are allowed to use calls $\begin{Bmatrix} \text{getl}(x) \\ \text{getr}(x) \end{Bmatrix}$ to boolean procedures, which attempt to read $\begin{Bmatrix} \text{lft} \\ \text{rht} \end{Bmatrix}$ and yield true if the input file is nonemepty, and false otherwise. If the input file is nonempty then there is a side effect: x becomes the first number in the file. A call put(x) transfers the first number x of one of the input files to the output file and advances both the input file concerned and the output file one position.

One solution is to envisage the merging process to consist of two stages. In the first stage both files are nonempty. The second stage begins as soon as at least one input file is empty (assertion $\neg sl \vee \neg sr$ below), because then all that remains to be done is to copy the entire remainder of the other input file onto the output so that both input files become empty (assertion $\neg sl \ \& \ \neg sr$ below). The following solution may well be a typical outcome of an exercise in Disciplined Programming. It clearly marks the two stages; the assertions marking these are literally the terminating condition for the corresponding do...od.

```
sl,sr:=getl(u), getr(v);
do sl & sr → if u ≤ v → put(u); sl:=getl(u)
             □ u ≥ v → put(v); sr:=getr(v)
             fi
od;
{¬sl ∨ ¬sr}
do  sl → put(u); sl:=getl(u)
 □  sr → put(v); sr:=getr(v)
od;
{¬sl & ¬sr}
```

The above algorithm performs superfluous operations in many situations. For example, in the first stage both input files are tested for nonemptiness, whereas it is only necessary to do so for the one from which a number has just been taken. If a few extra tests don't hurt, then we might as well use the following algorithm, which is closer to the Ultimate in Elegance.

```
sl,sr:=getl(u), getr(v);
do sl & (sr & u ≤ v ∨ ¬sr) → put(u); sl:=getl(u)
□ sr & (sl & u ≥ v ∨ ¬sl) → put(v); sr:=getr(v)
od
```

where the terminating condition, falsity of all guards, simplifies to ¬sl & ¬sr.

Let us now construct by stepwise aggregation a program at the other extreme: no information will be thrown away and not a single superfluous test will be tolerated in the result.

The state is determined by the values of lft, rht, and tpt. The input specification is

$$\text{lft} = \text{lft}_o \;\&\; \text{rht} = \text{rht}_o \;\&\; \text{tpt} = \phi$$

where lft_o and rht_o are arbitrary, given, sorted files and ϕ is the empty file.

The output specification is

$$\text{tpt} = \text{merge}\ (\text{lft}_o, \text{rht}_o)$$

An intermediate assertion is again obtained by a judicious decomposition of the output assertion H as the conjunction of H_1 and H_2

$$H_1:\ \text{merge}\ (\text{lft},\text{rht}) \mathrel{<>} \text{tpt} = \text{merge}\ (\text{lft}_o, \text{rht}_o)$$
$$H_2:\ \text{lft} = \phi \;\&\; \text{rht} = \phi$$

where <> is the operation of appending one operand to the other. The algorithm tries to achieve H_1 and H_2 by repeatedly taking a step towards H_2 under invariance of H_1. This suggests using as counter the sum of the lengths of lft and rht.

Because we want to retain all information concerning the status of lft and rht, we will use several assertions each having the form of a conjunction of H_1 together with an assertion putting a constraint on lft and rht. Let us use the shorthand $\{\alpha,\beta\}$ for H_1 in conjunction with an assertion stating that lft has the form α and rht has the form β. For α or β we may have

?	stating that the file is possibly empty
$x{:}\alpha$	stating that the file is nonempty and that, moreover, x is the first number and that the remaining file has the form α
ϕ	stating that the file is empty

Box 7.1 shows in terms of verification conditions the properties of the commands.

$\{x{:}?,\beta\}$ put(x) $\{?,\beta\}$

$\{?,\beta\}$ getl(x) $\{x{:}?,\beta\}$

$\{?,\beta\}$ $\neg$getl(x) $\{\phi,\beta\}$

$\{\alpha,?\}$ getr(x) $\{\alpha,x{:}?\}$

$\{\alpha,?\}$ $\neg$getr(x) $\{\alpha,\phi\}$

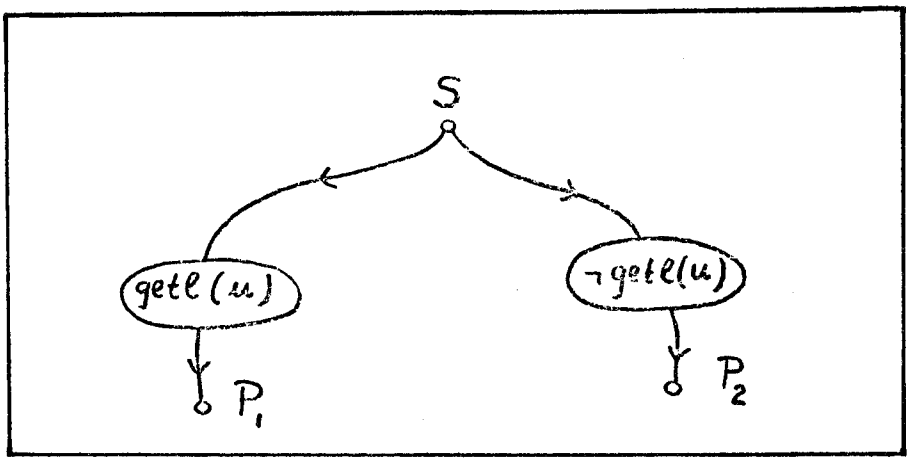
Box 7.1

The problem of finding the merge algorithm is now to get from $\{?,?\}$ to $\{\phi,\phi\}$.

The counter is decreased by a call to put. This can only be done under invariance of H_1 if $\{u:?,v:?\}$, $\{u:?,\phi\}$, or $\{\phi,v:?\}$ hold, and then it must be done. In the remaining cases it is for at least one of the files unknown whether it is empty, and then the appropriate choice of getl or getr is called for. We therefore have initially the following verification conditions.

$$S \stackrel{df}{=} \{?,?\}\ \text{getl}(u)\ \{u:?,?\} \stackrel{df}{=} P_1$$

$$S \stackrel{df}{=} \{?,?\}\ \neg\text{getl}(u)\ \{\phi,?\} \stackrel{df}{=} P_2$$



Box 7.2

Now there are no arcs going out from nodes P_1 and P_2. In order to avoid failed computations we must add such arcs. We already know that the commands in those arcs must resolve the "?" in the right-hand position. We therefore add

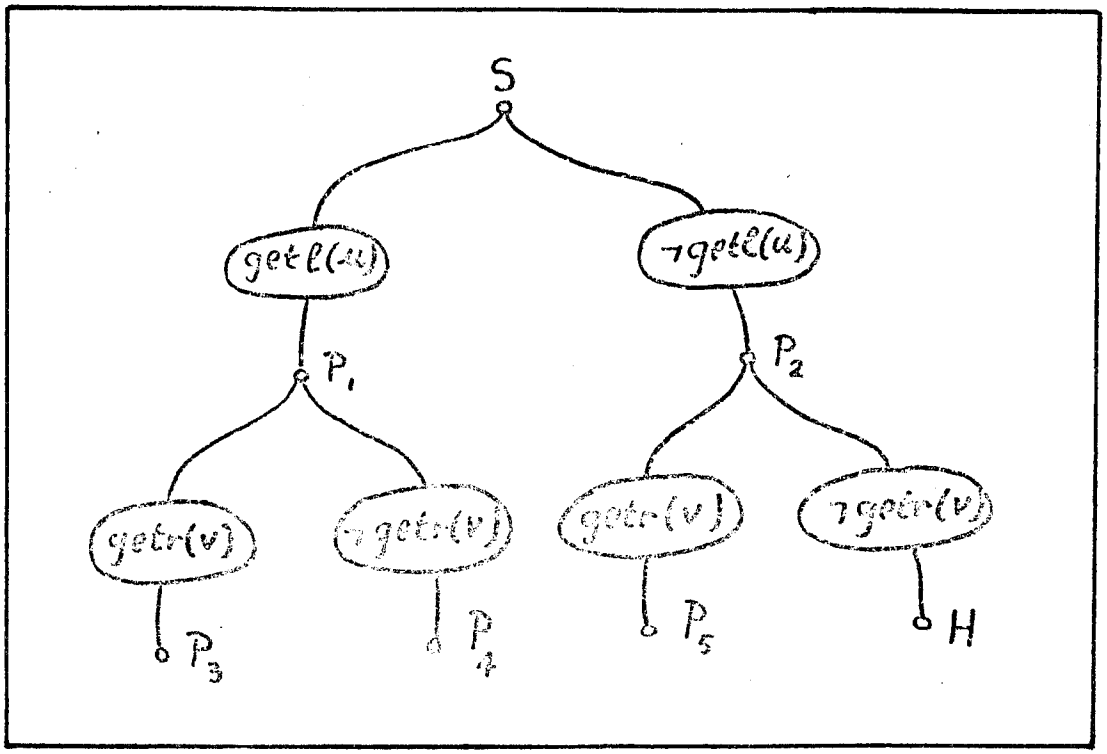
$$P_1 = \{u{:}?,?\}\ \ getr(v)\ \ \{u{:}?,v{:}?\} \stackrel{df}{=} P_3$$

$$P_1 = \{u{:}?,?\}\ \neg\, getr(v)\ \ \{u{:}?,\phi\} \stackrel{df}{=} P_4$$

$$P_2 = \{\phi,?\}\ \ getr(v)\ \ \{\phi,v{:}?\} \stackrel{df}{=} P_5$$

$$P_2 = \{\phi,?\}\ \neg getr(v)\ \ \{\phi,\phi\} \stackrel{df}{=} H$$

and get the flowgraph in Box 7.3.



Box 7.3

We have now introduced even more "dangling nodes", namely P_3, P_4, and P_5. It is determined by our heuristic which commands label the arcs going out from them:

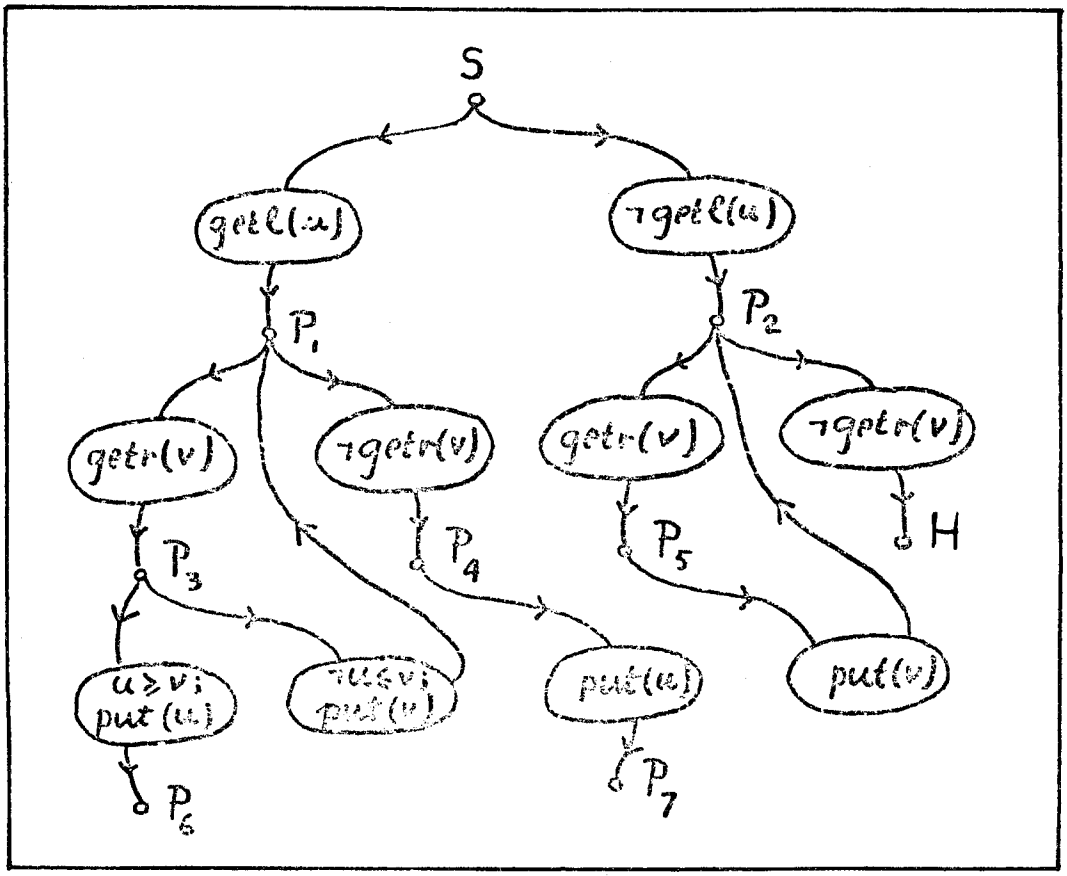
$P_3 = \{ u:?,v:? \}\ (u \leq v);\ put(u)\ \{ ?,v:? \} \stackrel{df}{=} P_6$

$P_3 = \{ u:?,v:? \}\ \neg(u \leq v);\ put(v)\ \{ u:?,? \} = P_1$

$P_4 = \{ u:?,\phi \}\ put(u)\ \{ ?,\phi \} \stackrel{df}{=} P_7$

$P_5 = \{ \phi,v:? \}\ put(v)\ \{ \phi,? \} = P_2$

See Box 7.4.



Box 7.4

For the first time now we have not introduced more dangling nodes than we eliminated; in fact, only P_6 and P_7 remain. We now add:

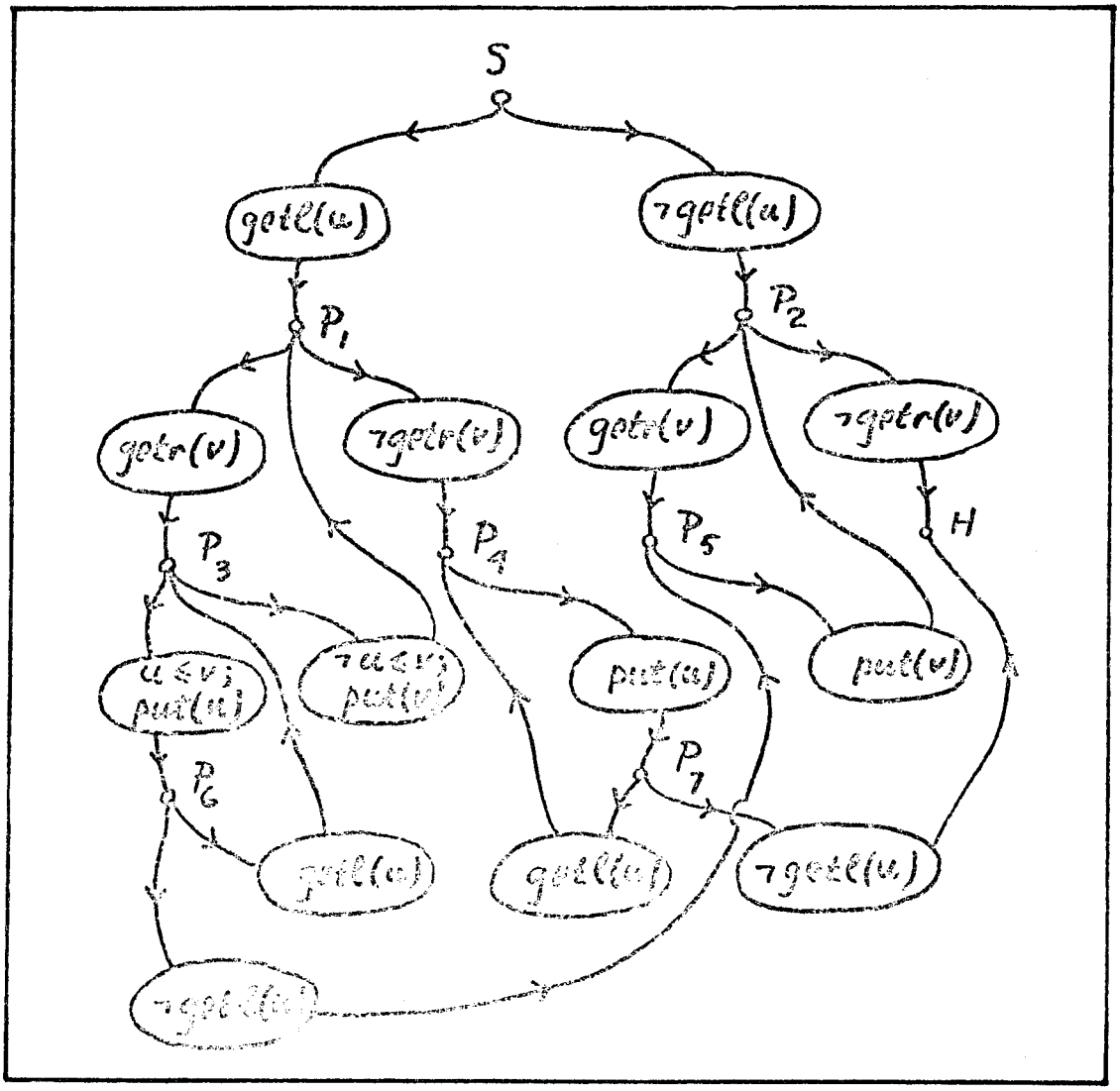
$P_6 = \{ ?, v{:}? \}\ \text{getl}(u)\ \{ u{:}?, v{:}? \} = P_3$

$P_6 = \{ ?, v{:}? \} \neg \text{getl}(u)\ \{ \phi, v{:}? \} = P_5$

$P_7 = \{ ?, \phi \}\ \text{getl}(u)\ \{ u{:}?, \phi \} = P_4$

$P_7 = \{ ?, \phi \} \neg \text{getl}(u)\ \{ \phi, \phi \} = H$

This time no dangling nodes have been introduced. In other words, there are no failed computations. It is easily checked that the arcs with a "put" command are a basis set, so there are no infinite computations. The flowgraph in Box 7.5 is therefore totally correct.



Box 7.5

The translation from a set of verification conditions to an Algol 60 program, as given with example I, is only one of several useful translations. In this example we translate a pair of verification conditions such as

$$S = \{?,?\}\ \text{getl}(u)\ \{u{:}?,?\} = P_1$$
$$S = \{?,?\}\ \neg\text{getl}(u)\ \{\phi,?\} = P_2$$

to

$$S:\ \underline{\text{if}}\ \text{getl}(u)\ \underline{\text{then goto}}\ P_1\ \underline{\text{else goto}}\ P_2$$

In this way the set of verification conditions of this example translate to the set of statements in Algol-60 shown in Box 7.6.

S: if getl(u) then goto P_1 else goto P_2;
P_1: if getr(v) then goto P_3 else goto P_4;
P_2: if getr(v) then goto P_5 else goto H;
P_3: if u ≤ v then begin put(u); goto P_6 end else begin put(v); goto P_1 end;
P_4: put(u); goto P_7;
P_5: put(v); goto P_2;
P_6: if getl(u) then goto P_3 else goto P_5;
P_7: if getl(u) then goto P_4 else goto H

Box 7.6

Note that the verification conditions are an unordered set. Written in any order they define the same flowgraph, which uniquely defines a set of computations. The ordering of the verification conditions has no meaning. Therefore the statements resulting from translation of verification conditions, such as those in the sub-boxes of Box 7.6, can be re-ordered without affecting the meaning of the resulting statement, provided that they are immediately preceded by

goto S;

and followed by the labelled dummy statement

H:

We will take advantage of the reorderability of the subboxes in Box 7.6 by making use of the programming languages's default transfer of control to the next statement, thus saving some jumps. Also, some tests have been inverted to create more opportunities for such an optimization.

No attempt has been made to obtain a result which is optimal with respect to program size. In fact, we cut out with scissors the boxed statements of Box 7.6, shuffled them around a bit, and then deleted unnecessary jumps and labels, sometimes after inverting a test. Box 7.7 is the result. Note that it is *irrelevant* whether this Algol code is understandable (we happen to think it isn't). Understandability is provided by the verification conditions in their historical development, with commentary, as given above. Correctness is guaranteed by the way the statements in Box 7.7 have been obtained from the verification conditions by translation and optimization.

```
S:  if ¬getl(u) then goto P2;
P1: if ¬getr(v) then goto P4;
P3: if u > v then begin put(v); goto P1 end;
    put(u);
    if getl(u) then goto P3;
P4: put(u); goto P7;
P2: if ¬getr(v) then goto H;
    put(v); goto P2;
P7: if getl(u) then goto P4;
H:
```

Box 7.7

There are no duplicated tests in the program in Box 7.7 because after executing an action, execution is already at, or transfers to, the program point associated with an assertion containing all information in the action's postcondition.

8. Concluding remarks

An incomplete understanding of programming with verification conditions may give rise to the following objections

a) the resulting programs exhibit *no structure*

b) the resulting code is *unreadable*

c) the method requires *no discipline* (hence must be sinful)

As for the first objection, let us go back to "structured programming", the harmfulness of goto's, and all that. Dijkstra [2] emphasised 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 simplify and accurately reflect the flow of control.

Notice that these considerations are relevant *only* in situations where sequence control has to be managed by the programmer. Our method consists of two stages. In the first stage assertions and verification conditions are collected until a flowgraph is obtained which is totally correct and translatable to Algol. In this stage sequence control need not, and should not, be considered. The next stage consists of automatable applications of translation and optimization rules. It is only here that sequence control appears: automatically, guaranteed correct, and, we have to add, almost entirely in the form of goto's. But even here sequence control should not occupy the programmer: if there is anything for him to do, it is to apply

the translation and optimization rules. The translation rules guarantee that just before executing a goto a certain assertion holds and that the goto then transfers to a label associated with that assertion: there is no harm in a goto if one knows where one is going to.

We conclude that the aspect of "structured programming" dealing with intellectual manageability of sequence control has become irrelevant now that there exist sufficiently systematic programming methods, such as in Dijkstra's own recent work [3] and in our method.

The objection of unreadability is dismissed simply by pointing out that the Algol programs resulting from our method are only meant for the compiler to be read, not for the human programmer. It is the set of verification conditions that is meant for reading, and indeed have to be read when checking their validity. Moreover, the rules we have given for checking the absence of failed computations are ascertainable from the flowgraph representation of the verification conditions without reference to sequence control.

The only constraint on the structure of the flowgraph which we find useful is that it be easy to find a basis set. A property of "structured" sequencing primitives is that the program text itself implies a basis set in the flowgraph corresponding to the program. We see no merit in this coupling of text and basis set. Consider for example the fairly complex flowgraph in Box 7.5. Even there it is easy to see that there is no infinite path from S without infinitely many actions put(u) or put(v). In other words, that put(u) and put(v) "cut all loops".

We conjecture that the phenomenon of a high concentration of complexity is unavoidable for algorithms avoiding superfluous tests. The stepwise aggregation aspect of programming with verification conditions gives a satisfactory way of constructing such algorithms.

It has been remarked that programming with verification conditions necessarily generates "structured" programs in disguise. Even if true we maintain that our method has the advantages of simplicity and flexibility as discussed in section 6. However, we neither know nor care whether the remark is justified: what is new here is that "structure" has become irrelevant.

"Think before you do" applies in programming no less than in other activities. In programming this injunction can be followed by preferring *reasoning* about assertions to *manipulating* program components. Perhaps the fundamental merit of programming with verification conditions is that assertions are central, rather than program components. Our method restores in full the flexibility of the unrestrained use of goto statements, yet maintains, and improves upon, the security and problem-solving power of "structured" programming.

9. Acknowledgements

We have profited from discussions with Peter Roosen-Runge of York University in Toronto and Keith Clark of Queen Mary College in London. A grant from the Canadian National Research Council has provided support.

10. Literature

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

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

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

[4] M.H. van Emden: Unstructured systematic programming. Report CS-76-09, Dept. of Computer Science, University of Waterloo, 1976.

[5] M.H. van Emden: Verification conditions as programs. Automata, Languages, and Programming (S. Michaelson and R. Milner, eds.), Edinburgh University Press, 1976.

[6] M.H. van Emden: Relational equations, grammars, and programs. Proc. Conf. on Theoretical Computer Science, University of Waterloo, 1977.

[7] R.W. Floyd: Assigning meanings to programs. Proc. Symp. App. Math. Vol. XIX (J.T. Schwartz,ed.), American Mathematical Society, 1967.

[8] E.C.R. Hehner: do considered od: A contribution to the programming calculus. Report CSRG-75, Computer Systems Research Group, University of Toronto, 1976.

[9] C.A.R. Hoare: An axiomatic basis for computer programming. Comm. ACM 12 (1969), 576-581.

[10] B.W. Kernighan and P.J. Plauger: The Elements of Programming Style. McGraw-Hill, 1974.

[11] R.A. Kowalski: Algorithm = Logic + Control. Dept. of Computation and Control, Imperial College, 1977.

[12] Z. Manna: Mathematical Theory of Computation. McGraw-Hill, 1974.

[13] J.C. Reynolds: Programming with transition diagrams. Programming Methodology (ed. D. Gries), Springer, 1978.

[14] R. Waldinger: Achieving several goals simultaneously. Machine Intelligence 8 (E.W. Elcock and D. Michie, eds.), Ellis Horwood, Chichester, and John Wiley & Sons, New York, 1977.

PROGRAMMING
WITH
VERIFICATION CONDITIONS

M.H. van Emden

Research Report CS-77-35
Department of Computer Science
University of Waterloo

November 1977

Preface

In "Unstructured Systematic Programming" [4] I gave an exposition of programming with verification conditions without, however, giving more than a few cursory remarks comparing this method with others. As might have been expected, I did not get away with that. The present paper is an extensively revised version of "Unstructured Systematic Programming", containing in addition a more complete discussion of the comparisons mentioned above.

M.H. van Emden
Waterloo, 11 September 1977

Preface to the second edition

The program on page 42 in the first edition is wrong. At least, I discovered an error in the translation from the program on page 40 to the one on page 42. Such translations should be done as mechanically as possible, because the error was made in handcopying. In this edition, page 42 contains a corrected program for which the required copying has been done on a Xerox copier, so that at least this source of error has been eliminated.

M.H. van Emden
Waterloo, 4 September 1978

PROGRAMMING

WITH

VERIFICATION CONDITIONS

M.H. van Emden

ABSTRACT

This paper contains an exposition of the method of programming with verification conditions. Although this method has much in common with the one discussed by Dijkstra in "A Discipline of Programming", it is shown to have the advantage in simplicity and flexibility. The simplicity is the result of the method's being directly based on Floyd's inductive assertions. The method is flexible because of the way in which the program is constructed in two stages. In the first stage a set of verification conditions is collected which corresponds to a program in "flowgraph" form. In this stage sequencing control is of no concern to the programmer. Control is introduced in the second stage, which consists of automatable applications of translation and optimization rules, resulting in conventional code. Although our method has no use for the sequencing primitives of "structured programming", it is highly secure and systematic.

Index Terms: correctness-oriented programming, structured programming, verification, control structure, bottom-up programming, invariant assertions.

1. Introduction

The fact that programming is expensive and error-prone has been in the past decade a source of surprise, alarm, or even despair; it became the target of a large amount of activity now known as "software engineering". What is so special about programs that they give us that much trouble? Even if we can do no better than to say that it has to do with their complexity, then we can already distinguish two ways in which this phenomenon arises. In the first place there is the *amount* of complexity which large programs typically contain by virtue of sheer size. Then there is the possibility of *concentration* of complexity: even a very small program can be difficult to understand and to verify.

The first phenomenon, that of a large amount of complexity, may be tackled by the "top-down" method described by Dijkstra [2], which is based on a process of abstraction where the solution to the original problem is conceived as a simple algorithm using, possibly very powerful, actions of a virtual machine. These actions 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 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 actions are finally implemented on the available machine.

The top-down method 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 and that one should have within a single layer as great a chunk of the total amount of complexity as one can safely handle.

In other words, each action of a virtual machine should have as great an amount of complexity as one can safely handle without taking recourse to decomposition. To increase this amount we need a method that works in a direction opposite to the one of top-down programming, that is, we need "bottom-up" programming as well. The terminology of top-down versus bottom-up is not sufficiently informative, being based on the way we usually draw trees, which happens to be upside-down. What matters is that the top-down method *decomposes*, and that the complementary bottom-up method can be said to proceed by *aggregation*.

This paper contains an exposition of *programming with verification conditions* and a discussion of its methodological implications. The method allows decomposition in much the same way as other methods do; it is distinguished by the ease with which it allows step-wise aggregation under preservation of partial correctness. A discussion of other methodological aspects is postponed to sections 6 and 7 because we prefer to have an example to refer to. We review the required tools in sections 3 and 4, and give an example in section 5.

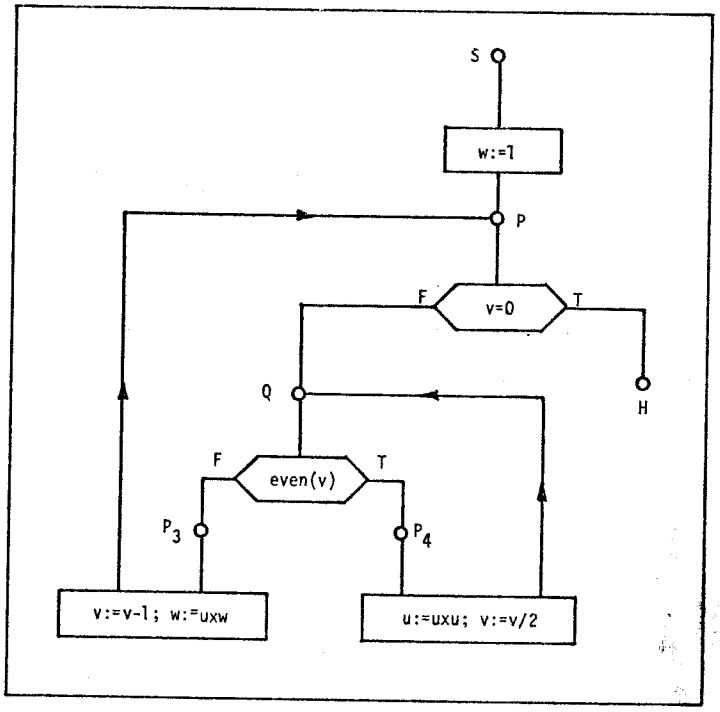
2. Related Work

Our method builds upon, simplifies, and extends Dijkstra's work [1,2,3]. Reynold's transition diagrams [13] are closely related to our flowgraphs. Hehner's paper [8] contains a critique of Dijkstra's do...od construct and some valuable hints on finding invariants. In [4] we gave a description and an example application of programming with verification conditions, but hardly any comparisons with other methods of systematic, correctness-oriented programming. In [5] we elaborated the relationship between programs and their verification conditions in the framework of first-order predicate logic. In [6] we characterized the meaning of both flowgraphs and verification conditions by means of minimal fixpoints.

Kowalski's paper [16] argues that it is possible and useful to decompose an algorithm into a "logic" component and a "control" component. One application of this decomposition is to explain the basic principle of our method: that a set of verification conditions is both a logical statement true independently of any sequence control, yet also a flowgraph program with a practically useful sequence control.

3. Flowcharts, Floyd's method, and flowgraphs

Consider the following example of a flowchart for an exponentiation algorithm.



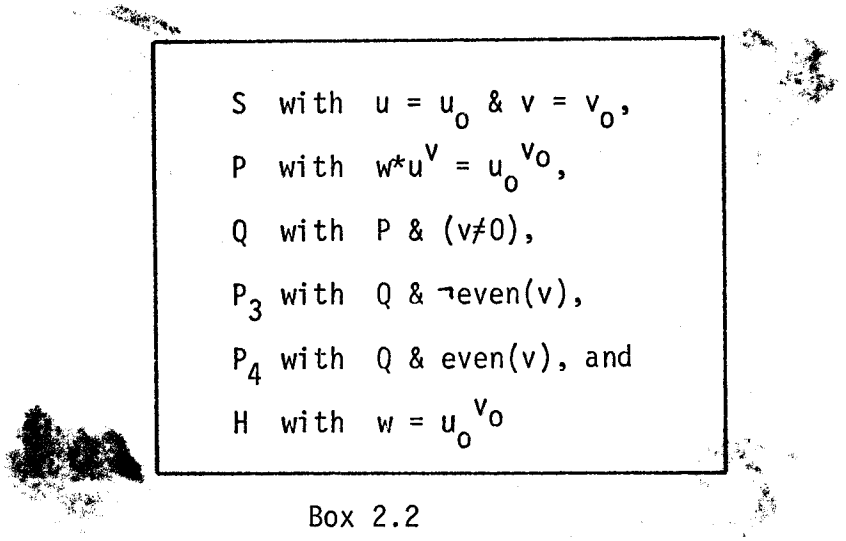
Box 2.1

The "labels" S,P,Q,P_3,P_4,H are names of program points. A "state" is determined by the contents of the registers u,v, and w. Execution of the flowchart can be regarded as the construction of a computation, which is, for the purposes of this paper, the sequence of (label,state)-pairs recording which labels are successively encountered during execution and, for each label, the corresponding state. For example,

$(P_3,(8,5,1))$, $(P,(8,4,8))$, $(Q,(8,4,8))$, $(P_4,(8,4,8))$, $(Q,(64,2,8))$

is a subsequence of a computation of the flowchart.

According to Floyd's method [7,12] of proving correctness, an "assertion" is associated with each label, with the intention of ensuring that whenever execution reaches a label, the state satisfies the associated assertion. For example, Box 2.2 lists useful associations.



Box 2.2

In Floyd's method, a flowchart is considered "verified" if execution reaching a label L with state s implies that the assertion associated with L is true of s. The proof that a flowchart is verified proceeds by induction. The basis of the induction is that the initial state satisfies the assertion associated with the start label, which can, in this example, always be satisfied by a suitable choice of u_0 and v_0. The induction step consists of proving a number of "verification conditions", namely statements of the form

{ precondition} action { postcondition}

where the precondition and the postcondition are assertions. The meaning is that

if state x satisfies "precondition" and
"action"transforms x to y
then state y satisfies "postcondition"

The verification conditions required for the flowchart in Box 2.1 are the following:

{S}	w:=1 {P}
{P}	successful exeuction of "v=0" {H}
{P}	successful execution of "v≠0" {Q}
{Q}	successful execution of "even(v)" {P_4}
{Q}	successful execution of "¬even(v)" {P_3}
{P_4}	u:=uxu; v:=v/2 {Q}
{P_3}	v:=v-1; w:=uxw {P}

Box 2.3

where the assertions are as given in Box 2.2. The induction of Floyd's method proves that execution reaching H implies that $w=u_o^{v_o}$, that is, the result of the exponentiation is found in w. The induction has not proved that execution *will* reach H, which is indeed not always true; for example not when $v_o < 0$.

A result of the form

> if the initial state satisfies assertion S
>
> and the flowchart terminates
>
> then the final state satisfies assertion H

is called *partial* correctness because termination is assumed, not proved. Note that a verification condition is itself a statement of partial correctness for an action. A result of the form

> if the initial state satisfies assertion S
>
> then the flowchart terminates
>
> and the final state satisfies assertion H

is called *total* correctness.

According to Floyd's method termination is proved separately from partial correctness by showing that in each possible computation an integer, which is bounded from below, is decremented a sufficient number of times for an infinite computation to be impossible. This general principle can be used in several different ways. We shall return to termination later and explore first the connection between programs and their verification conditions.

From a mathematical point of view, an action can be a binary relation between input states and output states. Expressed in terms of sets, such

a relation R is the set of pairs (x,y) such that (x,y) in R iff y is a possible state after executing the action represented by R, when x is the state before. We will admit indeterminate commands, so y is *a possible* output state rather than *the* output state. Mathematically speaking, we may have that $(x,y_1) \in R$ and $(x,y_2) \in R$ and $y_1 \neq y_2$.

Relations can express another important computational phenomenon: it may be that for some x there exists no y such that (x,y) in R. This expresses the fact that for some input states the action of a command is "not defined". For example, if the input state x is such that w = 0, then for the action u:=u/w there is no corresponding output state y.

It will be useful to have *a dummy action* (so called after Algol 60's "dummy statement"): an action which does not change the state. These are modelled by the identity relation I: $(x,x) \in I$ for every state x.

If a_1 and a_2 are actions (represented by the relations R_1 and R_2), then $a_1;a_2$ is the action obtained by first executing a_1 and then a_2. The action $a_1;a_2$ is represented by the product of R_1 and R_2: (x,y) is in the product iff $(x,z) \in R_1$ and $(z,y) \in R_2$ for some state z.

Because every relation is a set and every set of pairs of states is a relation, every subset of a relation is a relation again. We are especially interested in relations which are subsets of the identity relation, such as

$$\{ (x,x) \mid B \text{ is true in state } x\}$$

where B is some assertion.

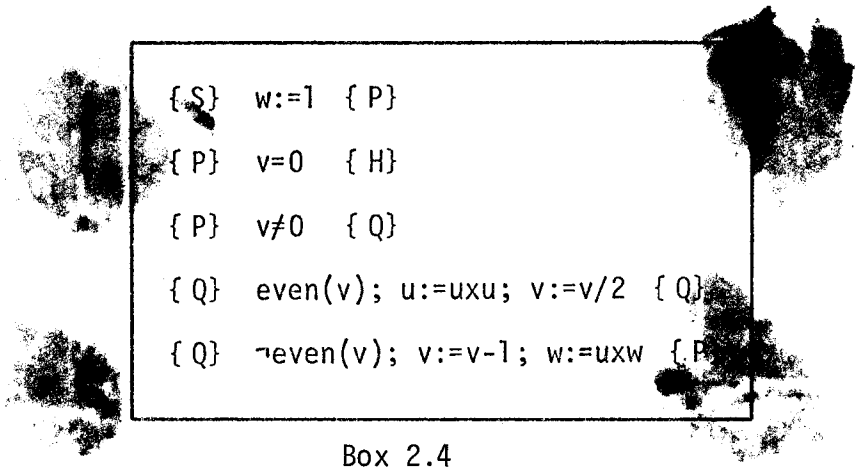
An action which is represented by such a subset of the identity is called a *guard*. If the input state to a guard satisfies the assertion, then the output state exists and is the same; otherwise no output state exists. If we write an assertion where one expects an action, then a guard is meant. For example

$$(v:=v/2);\ \text{integer}(v)$$

is an action and it has the same input-output behaviour as

$$\text{even}(v)\ ;\ (v:=v/2)$$

With these conventions we write the verification conditions of Box 2.3 as



Box 2.4

These verification conditions are more succinct than the ones in Box 2.3 and they no longer have the same direct relationship to the flowchart in Box 2.1. We now define independently of flowcharts a representation of sets of verification conditions as labelled directed graphs, which we call *flowgraphs*. These turn out to be very similar to flowcharts.

For every assertion in a set of verification conditions there is a node in the corresponding flowgraph. For each verification condition there is an arc in the flowgraph directed from the precondition node to the postcondition node, labelled with the action of the verification condition. Let us also suppose that specifications for programs are given in the form of an input assertion and an output assertion. The input assertion is an assertion which the initial state may be assumed to satisfy. The output assertion is an assertion which the final state must satisfy. The node in the flowgraph corresponding to the input assertion (output assertion) is called the start node (halt node). In this paper we use the letters S and H, respectively. We assume that no verification condition has the input assertion as post-condition or the output assertion as precondition.

The flowgraph in Box 5.5 is the one corresponding to the verification conditions in Box 2.4.

Although flowgraphs are defined as a graphical representation of verification conditions, they also define sets of computations, hence, they are algorithms. The execution of a flowgraph can be pictured as a token tracing a path from the start node to the halt node through the graph, with the following constraint: the token carries a state and it may only pass through an arc if the action labelling the arc is defined for the state. As a result of passing through the arc the state on the token is changed as required by the action.

For a more precise definition of computation we first define the *successor relation* among (node,state)-pairs: (N',x') is a successor of (N,x) iff

there is an arc from N to N' and (x,x') is in the action labelling this arc. A finite computation is a finite sequence of (node,state)-pairs.

$$(N_o,x_o),\ldots,(N_k,x_k)$$

such that N_o is the start node

and (N_{i+1},x_{i+1}) is a successor of (N_i,x_i) for $i=0,\ldots,k-1$

and (N_k,x_k) has no successor.

If N_k is the halt node then the computation is *successful*, otherwise it is *failed*; x_o is the *start state* of the computation.

An infinite computation is an infinite sequence of (node,state)-pairs

$$(N_o,x_o)\ ,\ (N_1,x_1),\ldots$$

such that N_o is the start node

and (N_{i+1},x_{i+1}) is a successor of (N_i,x_i) for $i=0,1,\ldots$

Flowgraphs are more general than conventional programs in two respects. A (node,state)-pair may have more than one successor; in that case the flowgraph is *indeterminate*. A finite computation may be failed; because of this possibility *any set of verification conditions corresponds to some flowgraph*.

4. Correctness of flowgraphs

Any set of verification conditions corresponds to a flowgraph, and any flowgraph is a program in the sense that it defines a set of computations. Of course, the flowgraph may have, apart from successful computations, also failed or infinite ones; it may not even have any successful computations at all! This close correspondence between verification conditions and flowgraph programs is essential for the method described in this paper.

The following theorem shows that, if a flowgraph is viewed as a set of valid verification conditions, then it is this set that proves partial correctness according to Floyd's method.

Theorem: If, for a verified flowgraph, a (node,state)-pair (L,x) occurs in a computation with start state x_o satisfying the assertion associated with the start node, then x satisfies the assertion associated with L.

Proof: A flowgraph is verified when its verification conditions are valid. Let (N_i,x_i) and (N_{i+1},x_{i+1}) be successive pairs in a computation. By the definition of computation, $(x_i,x_{i+1}) \in C$, where C is the action labelling an arc from N_i to N_{i+1}. Let the corresponding verification condition be $\{P\}\ C\ \{Q\}$, where $P(Q)$ is the assertion associated with $N_i(N_{i+1})$. The validity of $\{P\}\ C\ \{Q\}$ implies that, if x_i satisfies P, then x_{i+1} satisfies Q □

Note that the theorem applies to computations of all kinds: successful, failed, infinite. A special case of the theorem for successful computations proves partial correctness with as precondition (postcondition) the assertion

associated with the start node (halt node). To prove in addition total correctness it is often useful to prove separately the absence of infinite computations and the absence of failed computations.

For a proof of the absence of infinite computations the following method (adapted from Floyd [7,12]) 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 "basis set" of paths is traversed in a computation; a set B of paths is a *basis set* if and only if each infinite path starting at the start node has infinitely many occurrences of paths in B.

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 successful, failed, 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 each state in any 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 basis set B 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 actions labelling the successive arcs of a path in B and where x is in the assertion labelling the initial node of the path. In other words, the paths of the basis set "cut all loops".

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 failed computation with a node N in its final pair if for every $x \in N$ there exists a y and an action 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 failed computations starting from S.

5. Example I: Systematic construction of an exponentiation algorithm

The method of stepwise aggregation may be summarized as follows. Given as initial approximation to the required program only the specification in the form of a precondition and a postcondition, we successively add assertions and valid verification conditions, using the heuristic that failed computations be extended in the next approximation under the constraint of maintaining partial correctness and not introducing infinite computations. A statically ascertainable criterion, in this case the conditions for total correctness in the previous section, determines when the process of program construction by stepwise aggregation has been completed.

In this example the problem is to raise a number u_o to the power of a nonnegative integer v_o. We assume that states are triples of contents of registers called u,v, and w. The specification calls for a program which is totally correct with respect to precondition

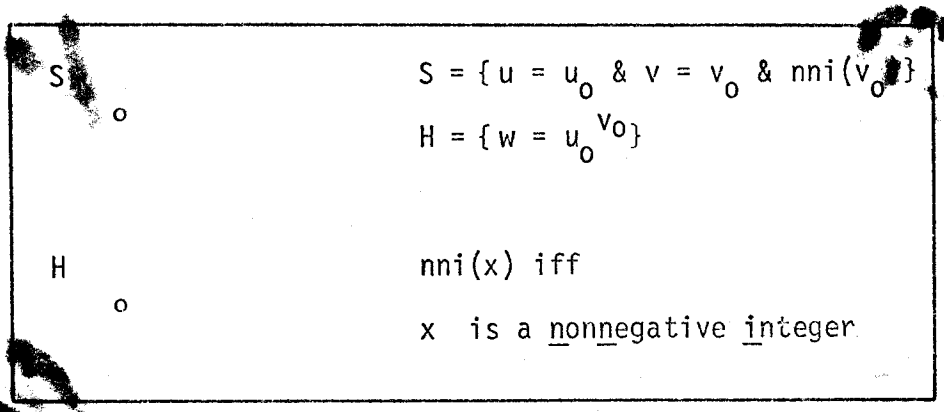
$$S = \{ u=u_o \ \& \ v=v_o \ \& \ nni(v_o)\},$$

where nni says whether its argument is a nonnegative integer, and postcondition

$$H = \{ w=u_o^{v_o}\}.$$

At this stage we have as approximation to the solution the following flowgraph, which is already partially correct. We maintain partial correctness with absence of infinite computations throughout the entire sequence of approximations,

and attempt to get closer to total correctness at each next step.



Box 5.1

The flowgraph in Box 5.1 needs improvement because it has no successful computations. Inserting a non-empty action X such that {S} X {H} may give an improvement, especially when X is $w:=u_0^{v_0}$, but let us assume that no such action is possible. Therefore at least one other node, with associated assertion, will have to be introduced. In order to obtain at least one successful computation, it is necessary that the assertion P be such that nonempty commands X and Y can be found such that {S} X {P} and {P} Y {H}.

As it turns out, the invention of the right P is almost all there is to invent in the algorithm ultimately obtained. If there would be a way of formally deriving P from S and H then it would be conceivable that the entire algorithm is constructed automatically. In the absence of such a formal derivation we assume P to be given as the assertion

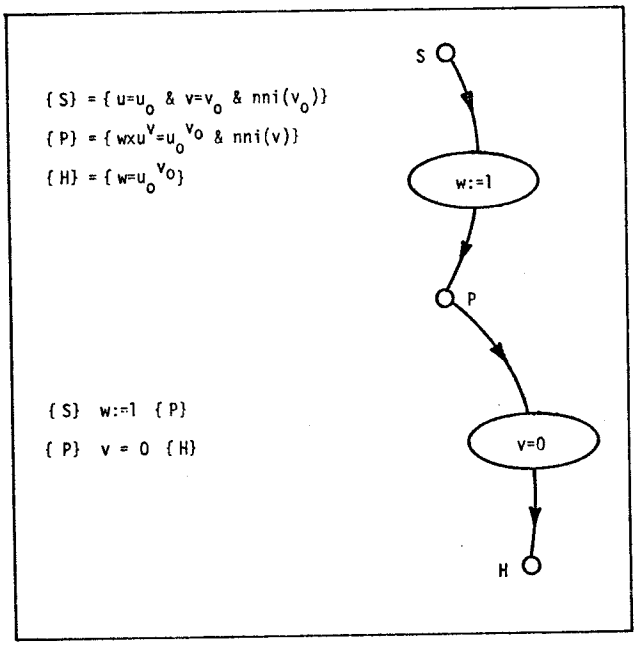
$$w \times u^v = u_o^{v_o}$$

as, for example, in Dijkstra's proof [3] of an exponentiation algorithm. We do not claim to have anything to contribute to solving the important problem of obtaining intermediate assertions. But even with this problem unsolved, reasoning in terms of assertions to find the required intermediate assertions is a much better method for constructing a correct program than directly manipulating program components, as seems to be the alternative. Therefore we will in this example bring some of the existing folklore [14,3,8] to bear upon the problem of at least making *plausible* the choice of P, so as hopefully to facilitate invention in future similar situations.

The fact that there is no useful X such that $\{S\}\ X\ \{H\}$ suggests that H is too hard to achieve directly. A commonly used heuristic is to *divide* (so as hopefully to *conquer*) into two easier goals, say H_1 and H_2. In easy cases, H_1 and H_2 are independent in the sense that H can be achieved by first achieving H_1 and then H_2. When H_1 and H_2 are not independent, achieving H_2 spoils the already obtained solution to H_1, and vice versa. But if we are lucky, and patient enough not to require H_2 to be achieved in a single step, we can take a *step towards* H_2 and then adjust the solution so that it still satisfies H_1. In that case we have the familiar iterative scheme where H_1 is an *invariant* and H_2 a stopping criterion.

The choice of P can be made somewhat plausible by regarding $\{H\} = \{w=u_o^{v_o}\}$ as a conjunction of two assertions $\{H_1\} = \{wxu^v = u_o^{v_o}\}$ and $\{H_2\} = \{v=0\}$.

H_1 can then be the intermediate assertion P which can be achieved initially by w:=1 and H_2 is achieved by passing the guard v=0. This gives not only as next approximation the flowgraph in Box 5.2, but it also suggests that v be used as counter for proving termination because a decrease in v is an obvious interpretation of "a step towards achieving H_2". Hence the conjunct nni(v) which henceforth appears in all nonhalt assertions.



Box 5.2

The flowgraph in Box 5.2 is again partially correct but only slightly less vacuous than the previous one: there are still not enough successful computations. We have not yet used the possibility of taking a step towards achieving H_2 under invariance of H_1. Let us first try a single command X satisfying $\{P\}\ X\ \{P\}$ and such that no infinite computations are introduced. For a simple proof of termination we try a basis set of single-arc paths, which must then include as a path the arc labelled X. The identities

$$wxu^v = w \times (uxu^{v-1}) = (wxu) \times u^{v-1}$$

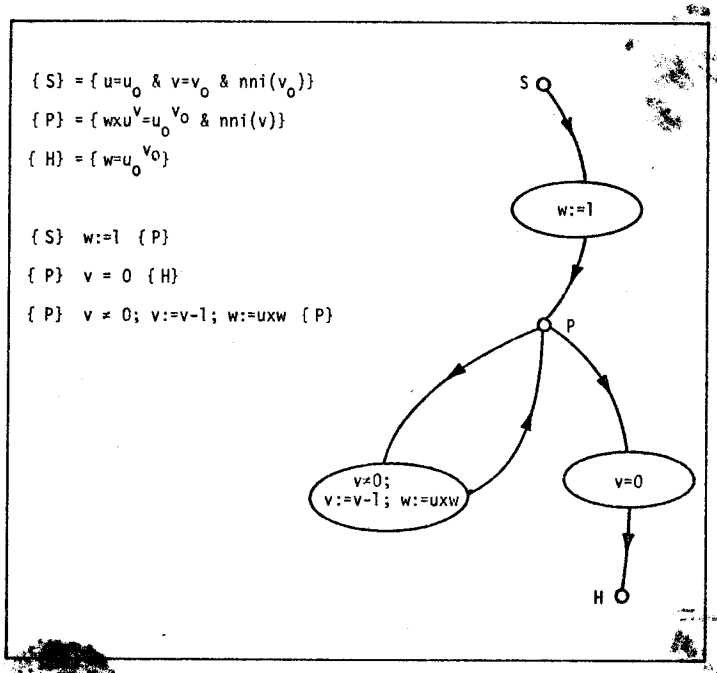
are useful for preserving P while decrementing v: they suggest trying for X the command v:=v-1; w:=wxu. However, in P the variable v has to be nonnegative; this must also hold after X. The straightforward way of ensuring this is to put

$$X = (v:=v-1;\ w:=uxw;\ v \geq 0) \qquad \ldots(5.1)$$

But actions with a guard following an assignment will give trouble in translating to a conventional language. So it is better to use the equivalent

$$X = (v > 0;\ v:=v-1;\ w:=uxw) \qquad \ldots(5.2)$$

Because P implies nni(v) we might as well take $v \neq 0$ instead of $v > 0$, as in the flowgraph in Box 5.3.



Box 5.3

The flowgraph in Box 5.3 is partially correct with respect to S and H, and has no failed or infinite computations starting from S. Efficiency can be increased by using the identity

$$u^v = (uxu)^{v/2}$$

which usually decreases v faster. This suggests that another arc from P to P be introduced, labelled by the action

u:=uxu; v:=v/2

The requirement that v remain integral is taken into account by elaborating the action to

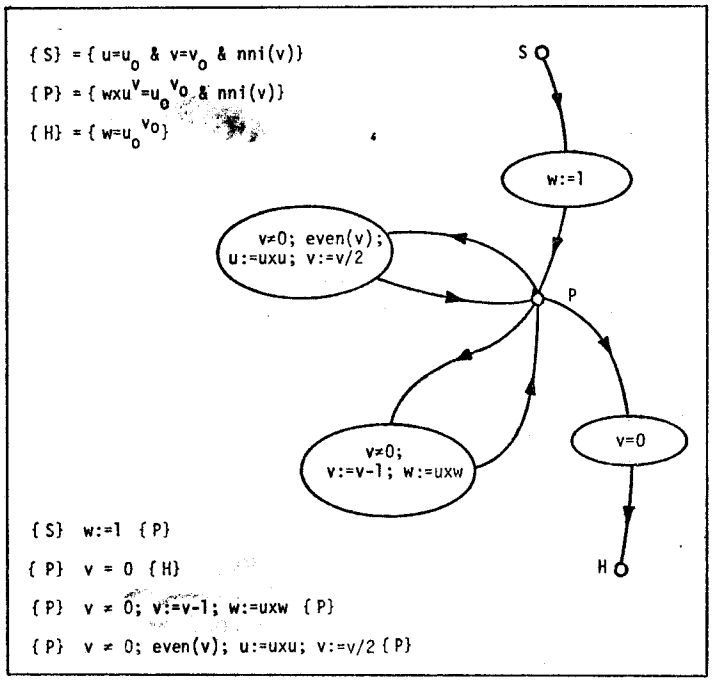
u:= u x u ; v:=v/2; integer (v) ...(5.3)

The desirability of having guards before assignments is the reason for using instead the equivalent action

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

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

v ≠ 0; even(v); u:=u x u; v:=v/2 ...(5.4)



Box 5.4

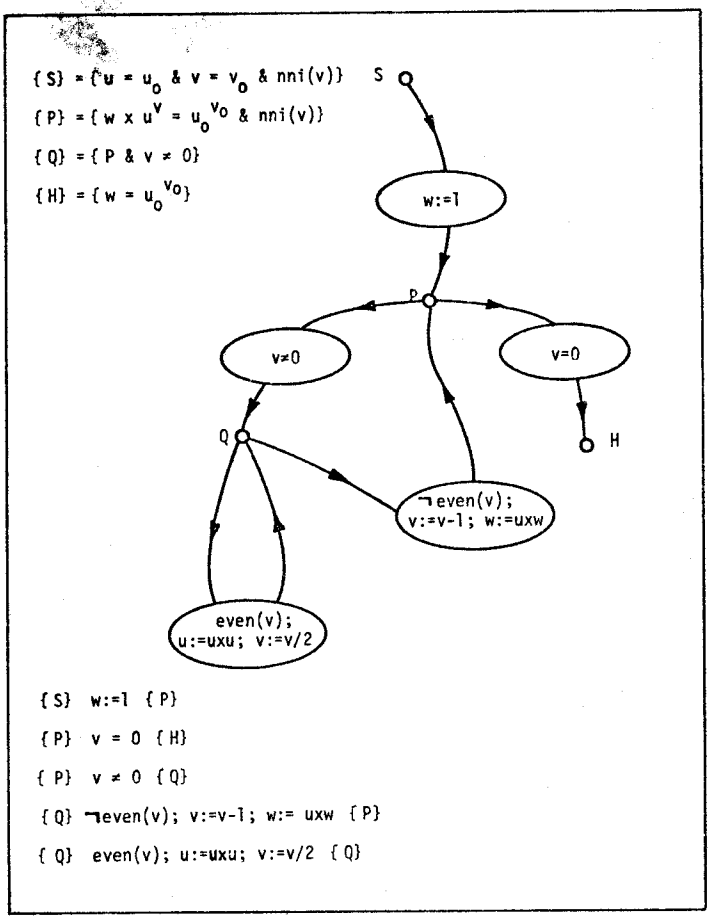
Note that the resulting flowgraph in Box 5.4 is indeterminate: efficient computations have been added to the set of computations of the previous flowgraph, but they have not replaced them. In order to exclude the inefficient computations it must be enforced that the usually more effective reduction in the counter is performed whenever possible. To achieve this the guard ¬even(v) is inserted into the action

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

to give

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

Two flaws remain: one is the fact that we have two actions beginning with the same guard $v \neq 0$ and starting from the same node P. An improved, equivalent flowgraph in Box 5.5 is obtained from the one in Box 5.4.



Box 5.5

The other flaw, which persists in Box 5.5, 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 activating the guard even(v) is superfluous. This situation is caused by the fact that

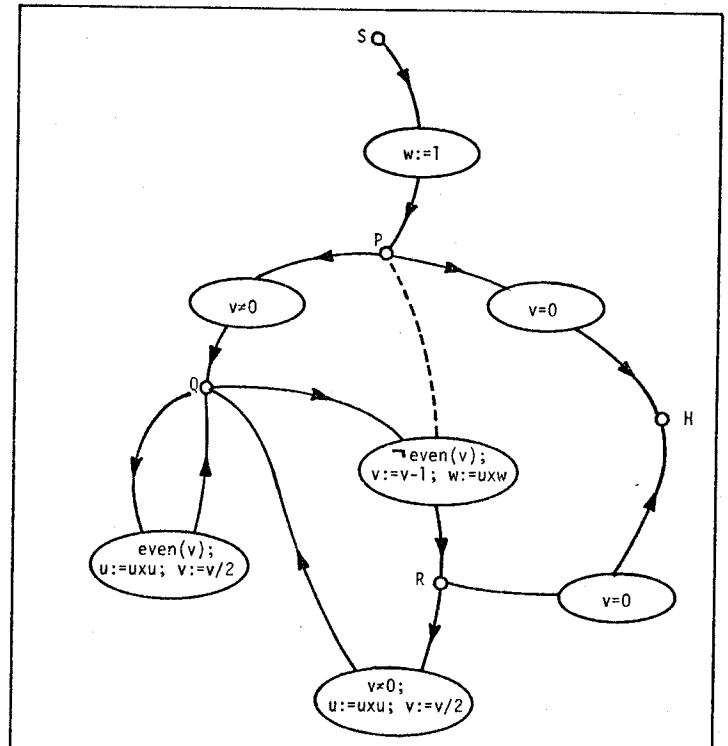
$$\{ Q\}\ \neg\text{even}(v);\ v := v-1;\ w := u \times w\ \{ P\}$$

can be replaced by the stronger verification condition

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

where $R = P\ \&\ \text{even}(v)$. 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 $\{ R\}\ v \neq 0;\ u := u \times u;\ v := v/2\ \{ Q\}$. It is only by including this last verification condition that we avoid the superfluous test. These changes give the flowgraph in Box 5.6 which is easily seen to be equivalent to the one in Box 5.7. Which of the two is preferred is largely a matter of taste.

All of the flowgraphs in Boxes 5.1,...,5.7 are partially correct and lack infinite computations. Those in Boxes 5.3, 5.5, 5.6, and 5.7 are determinate as well, lack failed computations, and their actions have a trivial translation to a conventional programming language. As an example, let us translate into Algol 60 the verification conditions of Box 5.7. The following translation rules are applicable for a useful class of verification conditions of which the flowgraphs must be without failed computations.



$\{S\} = \{u=u_0 \ \& \ v=v_0 \ \& \ nni(v)\}$

$\{P\} = \{wxu^v=u_0{}^{v_0} \ \& \ nni(v)\}$

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

$\{R\} = \{P \ \& \ even(v)\}$

$\{H\} = \{w=u_0{}^{v_0}\}$

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

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

$\{P\}$ v ≠ 0 $\{Q\}$

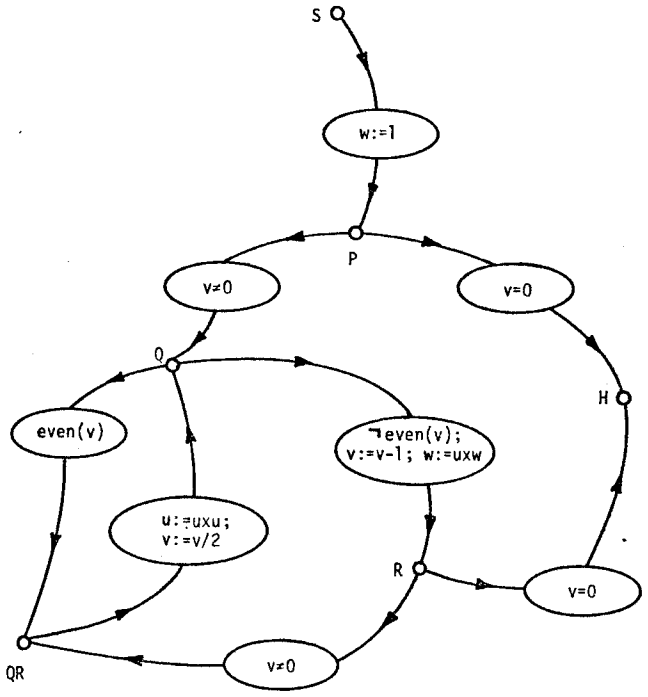
$\{Q\}$ even(v); u:=uxu; v:=v/2 $\{Q\}$

$\{Q\}$ ¬even(v); v:=v-1; w:=uxw $\{R\}$

$\{R\}$ v ≠ 0; u:=uxu; v:=v/2 $\{Q\}$

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

Box 5.6



$\{S\} = \{u=u_0 \ \& \ v=v_0 \ \& \ nni(v_0)\}$

$\{P\} = \{wxu^v = u_0^{v_0} \ \& \ nni(v)\}$

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

$\{R\} = \{P \ \& \ even(v)\}$

$\{QR\} = \{Q \ \& \ R\}$

$\{H\} = \{w = u_0^{v_0}\}$

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

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

$\{P\}$ v ≠ 0 $\{Q\}$

$\{Q\}$ ¬even(v); v:=v-1; w:=uxw $\{R\}$

$\{Q\}$ even(v) $\{QR\}$

$\{QR\}$ u:=uxu; v:=v/2 $\{Q\}$

$\{R\}$ v ≠ 0 $\{QR\}$

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

Box 5.7

Let the verification conditions be ordered in such a way that those with the same initial assertion are contiguous (and call the resulting subsequence a segment). Within a segment the order is insignificant. 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

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

translates to

$$P:\ \sigma_1;$$
$$\ldots$$
$$\sigma_k;$$

where the σ's are determined by the translation rules:

If C_i is a guard then σ_i is

$\underline{\text{if}}\ \mathcal{C}_i\ \underline{\text{then goto}}\ Q_i$

If C_i is an assignment then σ_i is

C_i; $\underline{\text{goto}}\ Q_i$

If C_i is a guarded assignment γ; α, then σ_i is

$\underline{\text{if}}\ \gamma\ \underline{\text{then begin}}\ \alpha;\underline{\text{goto}}\ Q_i\ \underline{\text{end}}$

For example, the verification conditions for the flowgraph in Box 5.7 translate to the following fragment of an Algol-60 program:

```
S:  w:=1; goto P;
P:  if v=0 then goto H;
    if v≠0 then goto Q;
Q:  if ¬even(v) then begin v:=v-1; w:=uxw; goto R
                      end;
    if even(v) then goto QR;
QR:  u:=uxu; v:=v/2; goto Q;
R:  if v≠0 then goto QR;
    if v=0 then goto H;
H:
```

The application of some simple optimization rules gives:

```
S:  w:=1;
    if v=0 then goto H;
Q:  if ¬ even(v) then begin v:=v-1; w:=uxw
                          ;   if v=0 then goto H
                      end;
    u:=uxu; v:=v/2; goto Q;
H:
```

6. A methodological discussion of example I

Dijkstra [1] argued that it is necessary to prove programs correct and he suggested that the difficulties encountered in attempts at program verification are caused by the peculiar approach where a program is completed first and a proof is attempted afterwards. He showed that it is possible and advantageous to develop a program and its proof in parallel. He argued that 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 showed that an algorithm can be developed more easily by reasoning about assertions than by manipulating program components. Here the proof comes, in a sense, *before* the program.

The basic proof method relevant here is due to Floyd. Subsequently Hoare [9] used Floyd's method in a formal system for proving partial correctness with a rule of inference for each basic construct of the programming language. Dijkstra [3] expressed correctness in terms of "predicate transformers". Where Hoare applies rules of inference to obtain partial correctness, Dijkstra manipulates expressions denoting assertions and obtains total correctness. Each construct of the programming language is characterized by a predicate transformer which is a functional combination of the predicate transformers of the constituents of the construct.

In contrast with the above, our method of programming with verification conditions is directly based upon Floyd's original method [7,12], bypassing

any subsequent elaborations. This directness is made possible by the use flowgraphs, a form of program identical to a set of verification conditions. Results from our method are as provably correct as those from Hoare's or Dijkstra's, but we need no counterpart for Hoare's rules of inference of Dijkstra's calculus of predicate transformers for dealing with language constructs, simply because what little of language constructs we use, enters only in the translation phase from a flowgraph already proved totally correct. In particular, we claim that the problem-solving power of Dijkstra's use of the "wdec" predicate can be achieved in an easier way by informal reasoning of which the transitions from (5.1) to (5.2) and from (5.3) to (5.4) are examples.

We have argued above that programming with verification conditions is a *simplification* with respect to previous comparable methods. We now argue that our method has a *larger scope*: it allows the "divide and conquer" principle to be applied in more dimensions than previous methods do. Let us first review various ways in which the principle is useful in programming.

In programming (and elsewhere) confusion results when one tries to do more than one thing at a 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; yet being able to do so depends on a suitable *decomposition* of the task at hand. The first decomposition is suggested by Kernighan and Plauger's maxim [10]: "get it right before you make it faster". And the goal of getting it right can again be decomposed, with similar advantages.

There is much to be said for getting the program right before one makes it determinate: see the flowgraph in Box 5.4 which is correct, but not determinate. This decomposition, the second, we owe to Dijkstra [3]. And we suggest a third decomposition, namely to get the program right even before making it do anything at all: the partially correct flowgraphs in Boxes 5.1 and 5.2 have no useful computations. Flowgraphs can have failed computations; they may even fail to have any successful ones. This degree of freedom makes programming by stepwise aggregation possible: our initial program is the vacuously partially correct one directly obtained from the input-output specification; in each step an assertion or verification condition is added with the purpose of allowing failed computations to continue on toward success. The addition must maintain partial correctness and must not introduce any infinite computation.

Each of the above decompositions is useful. Programming with verification conditions uniquely contributes the third decomposition, while being equally suitable as other methods for the first and second decompositions.

Dijkstra's sequencing primitives [3] are unsuitable for the third decomposition; this is because of the, in our view unfortunate, way termination for do...od is defined. Note that if...fi can give rise to failed computations (Dijkstra refers to failure as "abortion"). The do...od construct cannot, by (Dijkstra's) definition, fail: the criterion for successful termination is implicitly, by default, determined by the guards, as the conjunction of their negations. We have argued elsewhere [6] that an explicit criterion for success, independent of the guards, is an improvement for goal-directed programming. Our proposal in [6] would make Dijkstra's primitives more suitable for the third decomposition.

7. Example II: A trade-off between complexity and efficiency of a program

The algorithm described by the flowgraph in Box 5.5 can be expressed in Dijkstra's language [3] as

```
w:=1;
do  v ≠ 0  →  do even(v)  →  u,v:=uxu,v/2
              od;
              v,w:=v-1,uxw
od
```

It is significant that this algorithm is chosen rather than the more efficient one described by the flowgraph in Box 5.7. The latter is more complex in the sense of having more program points, which provide a sufficient "memory" for results of tests, so that no test needs to be duplicated.

In this section we will discuss the trade-off between having few program points (hence a simple algorithm which looks elegant in a "structured" language) and avoiding superfluous tests, which requires a certain minimum number of program points, in our method nodes of the flowgraphs. The fact that such a trade-off exists is typically swept under the rug in discussions promoting "structured" programming, where there seems to be a bias towards elegant algorithms performing superfluous tests.

We are aware that it is not usually worth the complexities of eliminating superfluous tests because anyway most programs are used only a few times (if at all). But in those rare cases where optimal efficiency *is* important, a systematic method must also help in discovering a satisfactory algorithm.

And the most important advantage of programming with verification conditions is that it allows one to cover fluently the entire spectrum between an efficient algorithm without duplicated tests and a simple algorithm of which the flowgraph has few nodes. The example discussed in this section is chosen because the spectrum is rather wide.

Let us consider the problem of merging two sorted input files of numbers (a left file called lft and a right file called rht) into a single, sorted output file called tpt. We are allowed to use calls $\begin{Bmatrix} \text{getl}(x) \\ \text{getr}(x) \end{Bmatrix}$ to boolean procedures, which attempt to read $\begin{Bmatrix} \text{lft} \\ \text{rht} \end{Bmatrix}$ and yield true if the input file is nonemepty, and false otherwise. If the input file is nonempty then there is a side effect: x becomes the first number in the file. A call $\{\begin{smallmatrix} \text{putl} \\ \text{putr} \end{smallmatrix}\}$ transfers the first number of $\{\begin{smallmatrix} \text{lft} \\ \text{rht} \end{smallmatrix}\}$ to tpt and advances $\{\begin{smallmatrix} \text{lft} \\ \text{rht} \end{smallmatrix}\}$ one position.

One solution is to envisage the merging process to consist of two stages. In the first stage both files are nonempty. The second stage begins as soon as at least one input file is empty (assertion $\neg$sl $\vee$ $\neg$sr below), because then all that remains to be done is to copy the entire remainder of the other input file onto the output so that both input files become empty (assertion $\neg$sl & $\neg$sr below). The following solution may well be a typical outcome of an exercise in Disciplined Programming. It clearly marks the two stages; the assertions marking these are literally the terminating condition for the corresponding do...od.

```
sl,sr:=getl(u), getr(v);
do sl & sr → if u ≤ v → put(u); sl:=getl(u)
             [] u ≥ v → put(v); sr:=getr(v)
             fi
od;
{¬sl ∨ ¬sr}
do  sl →  putl  ; sl:=getl(u)
 []  sr →  putr  ; sr:=getr(v)
od;
{¬sl & ¬sr}
```

The above algorithm performs superfluous operations in many situations. For example, in the first stage both input files are tested for nonemptiness, whereas it is only necessary to do so for the one from which a number has just been taken.

Let us now construct by stepwise aggregation a program at the other extreme: no information will be thrown away and not a single superfluous test will be tolerated in the result.

The state is determined by the values of lft, rht, and tpt. The input specification is

$$\text{lft} = \text{lft}_o \;\&\; \text{rht} = \text{rht}_o \;\&\; \text{tpt} = \phi$$

where lft_o and rht_o are arbitrary, given, sorted files and ϕ is the empty file.

The output specification is

$$\text{tpt} = \text{merge}\ (\text{lft}_o, \text{rht}_o)$$

An intermediate assertion is again obtained by a judicious decomposition of the output assertion H as the conjunction of H_1 and H_2

$$H_1:\ \text{merge}\ (\text{lft},\text{rht}) \mathbin{<>} \text{tpt} = \text{merge}\ (\text{lft}_o, \text{rht}_o)$$
$$H_2:\ \text{lft} = \phi \;\&\; \text{rht} = \phi$$

where <> is the operation of appending one operand to the other. The algorithm tries to achieve H_1 and H_2 by repeatedly taking a step towards H_2 under invariance of H_1. This suggests using as counter the sum of the lengths of lft and rht.

Because we want to retain all information concerning the status of lft and rht, we will use several assertions each having the form of a conjunction of H_1 together with an assertion putting a constraint on lft and rht. Let us use the shorthand $\{\alpha,\beta\}$ for H_1 in conjunction with an assertion stating that lft has the form α and rht has the form β. For α or β we may have

- ? stating that the file is possibly empty
- $x:\gamma$ stating that the file is nonempty and that, moreover, x is the first number and that the remaining file has the form γ
- ϕ stating that the file is empty

Box 7.1 shows in terms of verification conditions the properties of the commands.

$\{x:?,\beta\}$ putl $\{?,\beta\}$

$\{\alpha,x:?\}$ putr $\{\alpha,?\}$

$\{?,\beta\}$ getl(x) $\{x:?,\beta\}$

$\{?,\beta\}$ $\neg$getl(x) $\{\phi,\beta\}$

$\{\alpha,?\}$ getr(x) $\{\alpha,x:?\}$

$\{\alpha,?\}$ $\neg$getr(x) $\{\alpha,\phi\}$

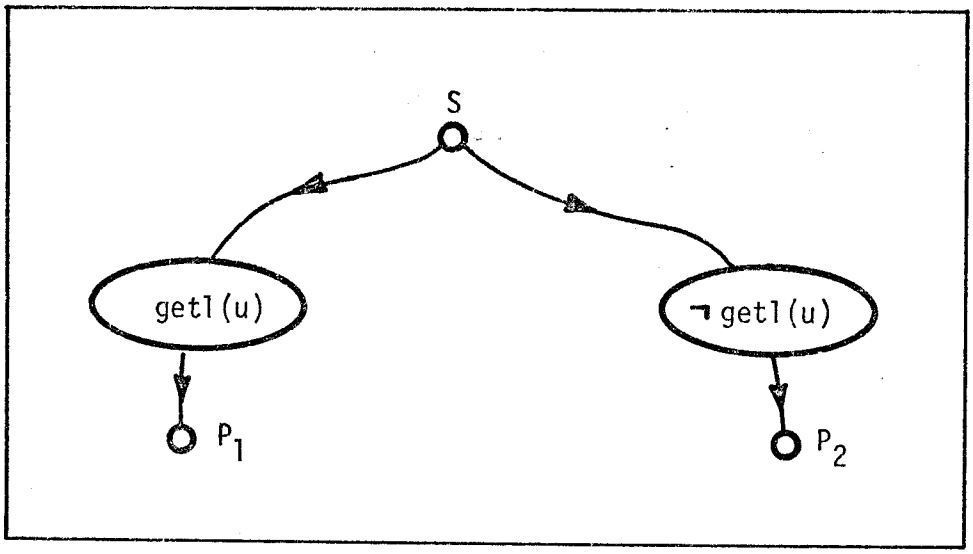
Box 7.1

The problem of finding the merge algorithm is now to get from $\{?,?\}$ to $\{\phi,\phi\}$.

The counter is decreased by a call to put. This can only be done under invariance of H_1 if $\{u:?,v:?\}$, $\{u:?,\phi\}$, or $\{\phi,v:?\}$ hold, and then it must be done. In the remaining cases it is for at least one of the files unknown whether it is empty, and then the appropriate choice of getl or getr is called for. We therefore have initially the following verification conditions.

$$S \stackrel{df}{=} \{?,?\}\ \text{getl}(u)\ \{u:?,?\} \stackrel{df}{=} P_1$$

$$S \stackrel{df}{=} \{?,?\}\ \neg\text{getl}(u)\ \{\phi,?\} \stackrel{df}{=} P_2$$



Box 7.2

Now there are no arcs going out from nodes P_1 and P_2. In order to avoid failed computations we must add such arcs. We already know that the commands in those arcs must resolve the "?" in the right-hand position. We therefore add

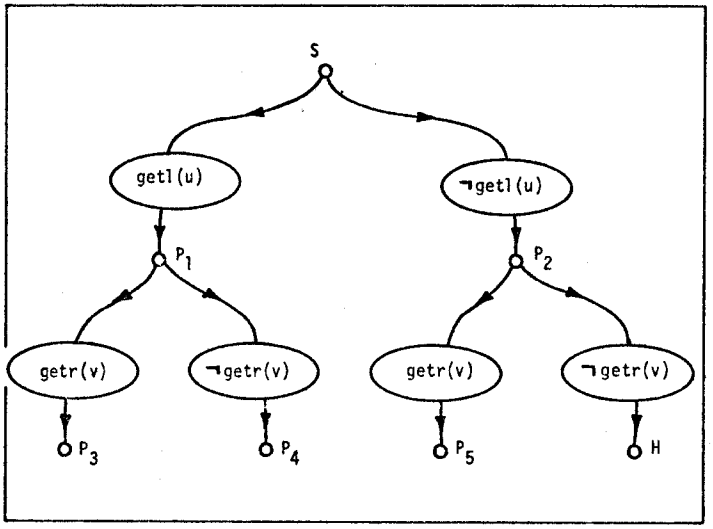
$$P_1 = \{u{:}?,?\}\ \ getr(v)\ \ \{u{:}?,v{:}?\} \stackrel{df}{=} P_3$$

$$P_1 = \{u{:}?,?\}\ \neg\, getr(v)\ \ \{u{:}?,\phi\} \stackrel{df}{=} P_4$$

$$P_2 = \{\phi,?\}\ \ getr(v)\ \ \{\phi,v{:}?\} \stackrel{df}{=} P_5$$

$$P_2 = \{\phi,?\}\ \neg getr(v)\ \ \{\phi,\phi\} \stackrel{df}{=} H$$

and get the flowgraph in Box 7.3.



Box 7.3

We have now introduced even more "dangling nodes", namely P_3, P_4, and P_5. It is determined by our heuristic which commands label the arcs going out from them:

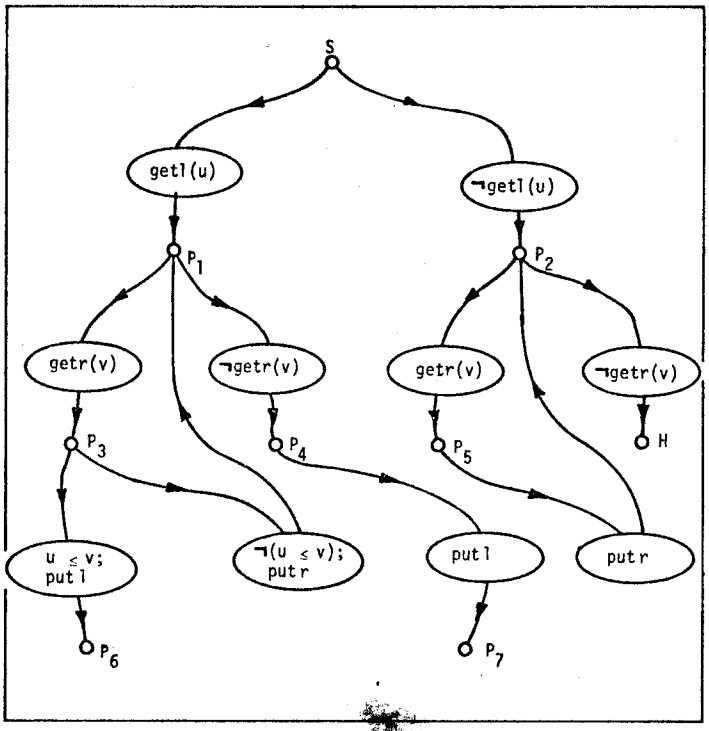
$P_3 = \{u:?,v:?\}\ (u \leq v);\ \textbf{putl}\ \{?,v:?\} \stackrel{df}{=} P_6$

$P_3 = \{u:?,v:?\}\ \neg(u \leq v);\ \text{putr}\ \{u:?,?\} = P_1$

$P_4 = \{u:?,\phi\}\ \textbf{putl}\ \{?,\phi\} \stackrel{df}{=} P_7$

$P_5 = \{\phi,v:?\}\ \text{putr}\ \{\phi,?\} = P_2$

See Box 7.4.



Box 7.4

For the first time now we have not introduced more dangling nodes than we eliminated; in fact, only P_6 and P_7 remain. We now add:

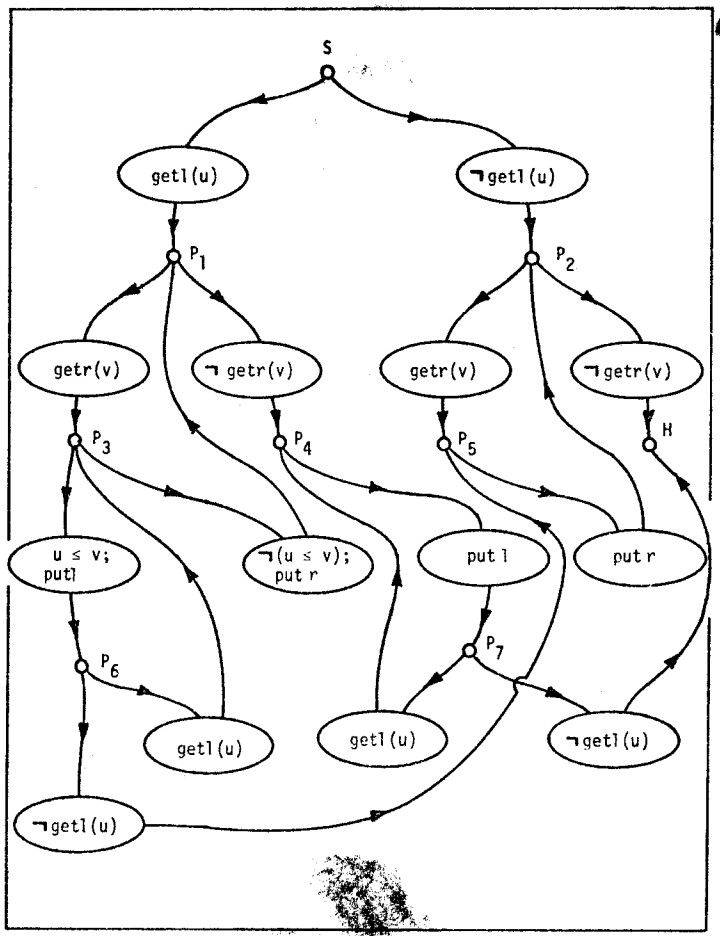
$P_6 = \{ ?,v:? \}\ \text{getl}(u)\ \{ u;?,v:? \} = P_3$

$P_6 = \{ ?,v:? \} \neg \text{getl}(u)\ \{ \phi,v:? \} = P_5$

$P_7 = \{ ?,\phi \}\ \text{getl}(u)\ \{ u:?,\phi \} = P_4$

$P_7 = \{ ?,\phi \} \neg \text{getl}(u)\ \{ \phi,\phi \} = H$

This time no dangling nodes have been introduced. In other words, there are no failed computations. It is easily checked that the arcs with a putl or putr command are a basis set, so there are no infinite computations. The flowgraph in Box 7.5 is therefore totally correct.



Box 7.5

The translation from a set of verification conditions to an Algol 60 program, as given with example I, is only one of several useful translations. In this example we translate a pair of verification conditions such as

$$S = \{?,?\} \quad \text{getl}(u) \quad \{u:?,?\} = P_1$$
$$S = \{?,?\} \quad \neg\,\text{getl}(u) \quad \{\phi,?\} = P_2$$

to

$$S: \ \underline{\text{if}}\ \text{getl}(u)\ \underline{\text{then goto}}\ P_1\ \ \underline{\text{else goto}}\ P_2$$

In this way the set of verification conditions of this example translate to the set of statements in Algol-60 shown in Box 7.6.

S: if getl(u) then goto P_1 else goto P_2;
P_1: if getr(v) then goto P_3 else goto P_4;
P_2: if getr(v) then goto P_5 else goto H;
P_3: if $u \leq v$ then begin putl; goto P_6 end else begin putr; goto P_1 end;
P_4: putl; goto P_7;
P_5: putr; goto P_2;
P_6: if getl(u) then goto P_3 else goto P_5;
P_7: if getl(u) then goto P_4 else goto H

Box 7.6

Note that the verification conditions are an unordered set. Written in any order they define the same flowgraph, which uniquely defines a set of computations. Therefore the statements resulting from translation of verification conditions, such as those in the sub-boxes of Box 7.6, can be re-ordered without affecting the meaning of the resulting statement, provided that they are immediately preceded by

goto S;

and followed by the labelled dummy statement

H:

We take advantage of the reorderability of the sub-boxes in Box 7.6 by utilizing the programming language's default transfer of control to the next statement, thus saving some jumps. Also, some tests are inverted to create more opportunities for such an optimization.

No attempt is made to obtain a result which is optimal with respect to program size. In fact, we cut out with scissors the boxed statements of Box 7.6, shuffled them around a bit, and then deleted unnecessary jumps and labels, sometimes after inverting a test. Box 7.7 is the result. Note that it is *irrelevant* whether this code is understandable. Understandability is provided by the verification conditions in their historical development, with commentary, as given above. Correctness is guaranteed by the way the statements in Box 7.7 are obtained from the verification conditions by translation and optimization.

```
S:  if getl(u) then goto P1;
P2: if ¬getr(v) then goto H;
P5: putr; goto P2;
P1: if ¬getr(v) then goto P4;
P3: if u > v then begin putr
                        ;     goto P1
                        end;
    putl;
    if getl(u) then goto P3
               else goto P5;
P4: putl;
    if getl(u) then goto P4;
H:
```

Box 7.7

There are no duplicated tests in the program in Box 7.7 because after executing an action, execution is already at, or transfers to, the program point associated with an assertion containing all information in the action's postcondition.

8. Concluding remarks

An incomplete understanding of programming with verification conditions may give rise to the following objections

a) the resulting programs exhibit *no structure*

b) the resulting code is *unreadable*

c) the method requires *no discipline* (hence must be sinful)

As for the first objection, let us go back to "structured programming", the harmfulness of goto's, and all that. Dijkstra [2] emphasised 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 simplify and accurately reflect the flow of control.

Notice that these considerations are relevant *only* in situations where sequence control has to be managed by the programmer. Our method consists of two stages. In the first stage assertions and verification conditions are collected until a flowgraph is obtained which is totally correct and translatable to Algol. In this stage sequence control need not, and should not, be considered. The next stage consists of automatable applications of translation and optimization rules. It is only here that sequence control appears: automatically, guaranteed correct, and, we have to add, almost entirely in the form of goto's. But even here sequence control should not occupy the programmer: if there is anything for him to do, it is to apply

the translation and optimization rules. The translation rules guarantee that just before executing a goto a certain assertion holds and that the goto then transfers to a label associated with that assertion: there is no harm in a goto if one knows where one is going to.

We conclude that the aspect of "structured programming" dealing with intellectual manageability of sequence control has become irrelevant now that there exist sufficiently systematic programming methods, such as in Dijkstra's own recent work [3] and in our method.

The objection of unreadability is dismissed simply by pointing out that the Algol programs resulting from our method are only meant for the compiler to be read, not for the human programmer. It is the set of verification conditions that is meant for reading. Moreover, the rules we have given for checking the absence of failed computations are ascertainable from the flowgraph representation of the verification conditions without reference to sequence control.

The only constraint on the structure of the flowgraph which we find useful is that it be easy to find a basis set. A property of "structured" sequencing primitives is that the program text itself implies a basis set in the flowgraph corresponding to the program. We see no merit in this coupling of text and basis set. Consider for example the fairly complex flowgraph in Box 7.5. Even there it is easy to see that there is no infinite path from S without infinitely many actions putl or putr . In other words, that putl and putr "cut all loops".

We conjecture that the phenomenon of a high concentration of complexity is unavoidable for algorithms avoiding superfluous tests. The stepwise aggregation aspect of programming with verification conditions gives a satisfactory way of constructing such algorithms.

It has been remarked that programming with verification conditions necessarily generates "structured" programs in disguise. Even if true we maintain that our method has the advantages of simplicity and flexibility as discussed in section 6. However, we neither know nor care whether the remark is justified: what is new here is that "structure" has become irrelevant.

"Think before you do" applies in programming no less than in other activities. In programming this injunction can be followed by preferring *reasoning* about assertions to *manipulating* program components. Perhaps the fundamental merit of programming with verification conditions is that assertions are central, rather than program components. Our method restores in full the flexibility of the unrestrained use of goto statements, yet maintains, and improves upon, the security and problem-solving power of "structured" programming.

9. Acknowledgements

Jack Alanen has helpfully and critically read an earlier draft, giving many valuable suggestions. We have profited from discussions with Peter Roosen-Runge and Keith Clark. The Canadian National Research Council has provided partial support.

10. Literature

[1] E.W. Dijkstra, "Concern for correctness as a guiding principle for program composition", The Fourth Generation (J.S.J. Hugo, ed.), Infotech, Maidenhead, 1971.

[2] E.W. Dijkstra, "Notes on structured programming", Structured Programming by O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Academic Press, 1972.

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

[4] M.H. van Emden, "Unstructured systematic programming", Report CS-76-09, Dept. of Computer Science, University of Waterloo, 1976.

[5] M.H. van Emden, "Verification conditions as programs", Automata, Languages, and Programming (S. Michaelson and R. Milner, eds.), Edinburgh University Press, 1976.

[6] M.H. van Emden, "Relational equations, grammars, and programs", in Proc. Conf. on Theoretical Computer Science, University of Waterloo, 1977.

[7] R.W. Floyd, "Assigning meanings to programs", in Proc. Symp. App. Math. Vol. XIX (J.T. Schwartz, ed.), American Mathematical Society, 1967.

[8] E.C.R. Hehner, "do considered od: A contribution to the programming calculus, Report CSRG-75, Computer Systems Research Group, University of Toronto, 1976.

[9] C.A.R. Hoare, "An axiomatic basis for computer programming", Comm. ACM, Vol. 12, pp. 576-581, 1969.

[10] B.W. Kernighan and P.J. Plauger, The Elements of Programming Style, McGraw-Hill, 1974.

[11] R.A. Kowalski, "Algorithm = Logic + Control", Dept. of Computation and Control, Imperial College, 1977.

[12] Z. Manna, Mathematical Theory of Computation, McGraw-Hill, 1974.

[13] J.C. Reynolds, "Programming with transition diagrams", Programming Methodology (ed. D. Gries), Springer, 1978.

[14] R. Waldinger, "Achieving several goals simultaneously", in Machine Intelligence 8 (E.W. Elcock and D. Michie, eds.), Ellis Horwood, Chichester, and John Wiley & Sons, New York, 1977.