Some Aspects of the Verification of Loop Computations

JAYADEV MISRA, MEMBER, IEEE

Abstract—The problem of proving whether or not a loop computes a given function is investigated. We consider loops which have a certain "closure" property and derive necessary and sufficient conditions for such a loop to compute a given function. It is argued that closure is a fundamental concept in program proving. Extensions of the basic result to proofs involving relations other than functional relations, which typically arise in nondeterministic loops, are explored. Several applications of these results are given, particularly in showing that certain classes of programs may be directly proven (their loop invariants generated) given only their input-output relationships. Implications of these results are discussed.

Index Terms—Inductive assertions, loop invariants, nondeterministic programs, program verification, proof rules, proving programs correct, proving program schemas.

I. Introduction

PROGRAM verification has now assumed an important role in the production of reliable software. The most popular method of verification, called *inductive assertion*, is based on the work of Floyd [10] as formalized by Hoare [14]. In this method, a set of predicates (assertions) is invented and attached to specific program points. These assertions are chosen in such a manner that they can be shown to hold whenever the program control reaches the corresponding point.

Programs that contain loops (as a majority do) pose the problem of inventing assertions that capture the dynamic relationship among the variables. The appropriate assertions must remain true (invariant) through successive iterations of the loop.

The results reported here grew out of an attempt to relate the result computed by a loop to the invariant. Clearly, different loops may transform their input data in different fashions. Surprisingly, however, certain general principles have emerged which relate the function computed by a loop to its invariant independent of the specific algorithm used for transformation.

Major results of this paper are the following.

1) A relationship is derived between the function computed by a loop and the invariant preserved by it. A loop computes a certain function F if and only if a certain proposition (involving F) is a loop invariant, and a boundary and a termination condition hold. This result holds whenever the loop meets a certain "closure" condition.

2) The form of the invariant is independent of the loop body, depending only on the given function F.

3) It is argued that "closure" is a fundamental notion in proving facts about loops.

4) These results are extended to prove general relations (other than functional relations) between input values and output results. It is shown that all equivalence relations can be proven in this manner. This is applied to proving facts about the nondeterministic loops of Dijkstra [5].

5) Two different program schemas are shown, which may be proven directly from their input-output specifications by a suitable extension of the basic theorem. These two program schemas occur often in programming.

Sufficient conditions for proving facts about loops have appeared in the literature [4], [9], [12], [13], [16], [22], [23], [25]. In particular, the notion of *subgoal induction* [23] has been found to be useful in a number of cases. However, since these conditions are not necessary, an incorrect program cannot be shown to be incorrect by these methods. Results in this paper provide necessary and sufficient conditions in order to relate the computed function to the invariant that must be proven.

The research reported here was motivated by the work of Mills [19] where the structure of computation of a loop program was related to the function it computes. The notion of "closure" was implicit in Mill's work.

The problem of termination is not discussed in this paper; termination has to be established by independent means. The basic theory is discussed in the next section, and some of its implications are studied in the following section. Some of these results have appeared in a number of places [1], [2], [20], [21]; this paper is an effort to present the results in a coherent framework.

II. Definition and Notation

Loops of the form "**while** B **do** S **enddo**" will be considered in this paper. The intuitive meaning is that the loop body S is not executed if B is false on entry; else S is executed one or more times until B becomes false. S is any one-in one-out program.

Some important parameters of the loop are the global and local variables, the input domain of values, and the output from the loop. These are discussed below.

Manuscript received August 13, 1976; revised March 30, 1978. This work was supported in part by NSF Grants DCR75-03749 and DCR75-09842 and Air Force Office of Scientific Research Contract F44620-71-00091.

The author is with the Department of Computer Science, University of Texas at Austin, Austin, TX 78712.

0098-5589/78/1100-0478$00.75

We call the following program schema $W(B, S)$:

begin
  Declarations for local variables t; {This is optional}
  while B **do** S **enddo**
end

Let x denote the vector of *values* of program variables that are external (global) to $W(B, S)$. Let x_0, x_f denote the initial and final values in x, respectively. We are not interested in the final values of local variables t. Initially, local variables have undefined values. The program may not access a local variable before it is assigned a value. Note that S may also have local variables of its own.

We make no restriction on the values of variables. In particular, variables may represent complex data objects such as graphs or entire data bases.

Normally, $W(B, S)$ would accept only those x satisfying some input condition D. D defines the domain of input to $W(B, S)$. We will interchangeably use "input assertion" and "input domain" to describe D. Let x_0 denote some initial input value to $W(B, S)$ and x_f the corresponding final value if $W(B, S)$ terminates. $W(B, S)$ *computes a function* F *on a domain* D if for every x_0 in D, $W(B, S)$ terminates and $x_f = F(x_0)$.

A proposition P is a *loop invariant* for $W(B, S)$ if $\{P \wedge B\}\ S\ \{P\}$, using the notation in [14]. An expression E is a *loop constant* if $(E = C)$ is a loop invariant for any arbitrary constant C. Thus, E's value following every iteration remains unchanged, and hence is equal to its initial value. An m-tuple of expressions $E = (e_1, e_2, \cdots, e_m)$ is a loop constant if $E = K$ is a loop invariant for any arbitrary m-tuple $K = (k_1, k_2, \cdots, k_m)$ of constants. Trivially, every loop invariant is a (Boolean) loop constant.

III. Loop Computation on Closed Domain

In order to prove that a program $W(B, S)$ computes a function F on a domain D, we need to find a proposition P such that

$D \Rightarrow P$ {Initially P is true}

$\{P \wedge B\}\ S\ \{P\}$ {P is a loop invariant}

$P \wedge \neg B \Rightarrow x_f = F(x_0)$ {x_0, x_f are initial and final values of the global variables}.

We show in this section that under a condition of "closure," P may be automatically generated from F.

Definition: The input assertion D is *closed* with respect to $W(B, S)$ if D is a loop invariant.

It follows from this definition that for a closed D, if D is true initially, it remains true through successive iterations, and hence it is true at termination.

The next theorem provides the necessary and sufficient conditions under which $W(B, S)$ computes a given function F on a domain D when D is closed with respect to $W(B, S)$.

The following lemma and its proof appear in Basu and Misra [1]. We present a stronger result in Theorem 1.

Lemma 1: Let D be closed with respect to $W(B, S)$. For every $x_0 \in D$, $W(B, S)$ computes F on D *if and only if* the following three conditions hold.

1) $W(B, S)$ terminates for every input x_0 in D.
2) $[D(x) \wedge \neg B(x)] \Rightarrow [F(x) = x]$.
3) $F(x)$ is a loop constant, i.e., $F(x)$ has a constant value through successive iterations of the loop.

Furthermore, conditions 1), 2), and 3) are mutually independent. □

Intuition behind the lemma is as follows. It is impossible for the loop to decide from the values of global variables x whether x is fresh input or whether x is the intermediate result with some other input. Hence, irrespective of whether x is an initial input or an intermediate result, the final value x_f must be identical. If input x_0 produces the sequence of intermediate values $x_1, x_2, \cdots, x_f$, then with input x_1, x_2, or any other x_i from the sequence, the same final value x_f must be obtained. $x_f = F(x_0) = F(x_1) = F(x_2) \cdots = F(x_f)$. Hence, $F(x)$ remains constant through successive iterations.

This argument has flaws, since local variables t might retain values from one iteration to the next. In such a case, they might provide some information as to whether the current x is a fresh input or some intermediate result. Conceivably, the program could make the distinction and produce different results in the two cases. However, the program cannot examine a local variable unless it is sure that x is not a fresh input. Otherwise, local variables have undefined values. It is possible to show that the local variables cannot effectively be examined unless they have received a value in the same iteration. It is thus impossible to carry information from one iteration to the next in local variables.

Thus, in either case, $F(x)$ is a loop constant. A simple observation is that for any input x for which B is false, $F(x) = x$ since the loop body will not be executed. This argument is the basis for a formal proof of the theorem that appears in [1].

One implication of this lemma is that the local variables of $W(B, S)$ can be made local to S without affecting the computation. These local variables can only be accessed if they have received values during the same iteration. Thus, without loss in generality, we can omit mention of such local variables or consider them as local variables of S.

Let $S(x)$ denote the values obtained by applying S to global variable values x (assuming undefined values of local variables). Lemma 1 can then be expressed in a more symmetrical form.

Corollary: Suppose D is a loop invariant for $W(B, S)$. $W(B, S)$ computes F over D if and only if the following conditions hold.

1) $W(B, S)$ terminates for every input x from D.
2) $[B(x) \rightarrow F(x) = F(S(x))] \wedge [\neg B(x) \rightarrow F(x) = x]$, $\forall x \in D$.

□

Example 1: Let $W(B, S)$ be the following program.

```
begin integer r, s;
  while v ≠ 0 do
    r ← v/2; {integer division}
    s ← 2 * r;
    if (s ≠ v) then w ← w * u endif;
    v ← r;
```

```
        u ← u * u
    enddo
end;
```

Suppose it is required to show that $W(B, S)$ computes $F\langle u, v, w\rangle = \langle g(u, v), 0, w * u^v\rangle$ on the domain $D = \{\langle u, v, w\rangle | v \geq 0, v \text{ integer}\}$ where $g(u, v) = u^{2^{\lceil \log_2 (v+1) \rceil}}$.[1]

We prove closure by showing that

$$[v \geq 0 \wedge v \neq 0] \Rightarrow [v/2 \geq 0].$$

We need to prove termination to satisfy condition 1). Finally, condition 2) tells us to prove the following propositions.

$$[v \neq 0] \rightarrow \begin{cases} g(u, v) = g(u^2, v/2) \text{ and } w * u^v \\ \qquad = w * u * (u^2)^{v/2}, \quad \text{if } v \text{ is odd} \\ g(u, v) = g(u^2, v/2) \text{ and } w * u^v \\ \qquad = w * (u^2)^{v/2}, \quad \text{if } v \text{ is even} \end{cases}$$

and $[v = 0] \rightarrow [g(u, v) = u \text{ and } w * u^v = w]$.

If these propositions are true, the loop computes the given function; otherwise it does not. □

This example illustrates one weakness of Lemma 1. We are interested only in the final values of w and v, and the function computed in the u component is of no interest, and hence may not be specified. Lemma 1, however, requires us to prove that each component remains a loop constant. If possible, we would like to show that only the function value computed in the w component is a loop constant. We have the following lemma, the proof of which is left as an exercise to the reader.

Lemma 2: Suppose $E = (e_1, e_2, \cdots, e_m)$ is a loop constant. Then each e_i is a loop constant. □

Note that the corresponding result for loop invariants does not hold; if $p \wedge q$ is a loop invariant, then possibly neither p nor q is a loop invariant. We can now combine Lemmas 1 and 2 to state a theorem that effectively says that, for the variables of interest, the computed function must be a loop constant. Let x denote the vector of all variable values, z a (single) variable value of interest, z_f its final value, and x_0, x_f the initial and final values, respectively, of all variables.

Theorem 1: Let D be closed with respect to $W(B, S)$. For every $x_0 \in D$, $z_f = g(x_0)$ if and only if the following three conditions hold.

1) $W(B, S)$ terminates for every input x_0 from D.
2) $[D(x) \wedge \neg B(x)] \Rightarrow [z = g(x)]$.
3) $g(x)$ is a loop constant. □

Note that $g(x)$ initially has the value $g(x_0)$. Hence, the value of $g(x)$ at any iteration must equal $g(x_0)$ and $g(x_f) = g(x_0)$. If $S(x)$ denotes the variable values obtained by executing S with values x, then Theorem 1 may be restated as follows.

Corollary: Under the conditions of Theorem 1 and assuming termination as in (1), $z_f = g(x_0)$ if and only if

$$[\neg B(x) \wedge D(x) \Rightarrow z = g(x)] \wedge [B(x) \wedge D(x) \Rightarrow g(x) = g(S(x))].$$ □

[1] $\lceil i \rceil$ = least integer greater than or equal to i.

If we are interested in several values on termination $z_{1f}, z_{2f}, \cdots, z_{rf}$, then Theorem 1 may be applied for each z_{if}.

Example 1 (continued): In order to show that

$$w_f = w_0 * u_0^{v_0}$$

it is only necessary to show that $w * u^v$ is a loop constant. □

We emphasize the following aspects of Theorem 1.

1) The conditions presented are necessary and sufficient. Hence, a loop program may be proved or disproved on the basis of this theorem.

2) The closure condition must be met before the theorem can be applied. We discuss the problem of nonclosed domains in a later section. This situation typically arises when the loop is preceded by an initialization.

3) The conditions are independent of the algorithm used for computing F. Hence, $F(x)$ is a loop constant, if the loop computes F, regardless of *how* F is computed.

IV. Extensions of Theorem 1

The three conditions in Theorem 1 are called the termination, boundary, and iteration conditions, respectively. The boundary condition is usually quite simple to derive and prove by considering those inputs for which the loop body is never executed. It is the iteration condition that is the central problem in dealing with loop structures. Theorem 1 completely solves the problem for closed domains when an explicit functional relationship between input and output is known. The next step is to relax one or both of these restrictions and look for iteration conditions similar to condition 3). The problem of nonclosed domains is discussed in a later section.

In this section, we assume that an explicit functional relationship between the input and output of $W(B, S)$ is not known. Instead, it is required to show that for some binary relation R, $x_f R x_0$ holds between initial value x_0 and final value x_f. For instance, it may need to be proven that the output is greater in magnitude than input. Of course, if we knew the functional relationship, we could prove such a fact easily. Often it is difficult, if not impossible, to guess the function being computed.

We characterize those relations for which a result analogous to Theorem 1 holds. Specifically, we show that if R is an equivalence relation and $x_f R x_0$ holds at termination, then $x R x_0$ must hold following every iteration. Conversely, if R is not such a relation (or some slight extension of it), then, in general, it is not possible to conclude that $x R x_0$ holds following every iteration. This result effectively characterizes the entire set of relations for which results analogous to Theorem 1 hold.

Theorem 2: Suppose D is closed with respect to $W(B, S)$. Let R be any equivalence relation on D such that every pair of initial and final values x_0, x_f belong to the same equivalence class under R. Then every intermediate result x arising out of initial x_0 must belong to the same equivalence class as x_0. □

A proof and a number of generalizations of this theorem appear in [20]. In particular, it is shown that this theorem can be extended to any R that is a slight generalization of equiva-

lence relations; furthermore, the theorem does not hold for any other relation.

The iteration condition [condition 3)] of Theorem 1 is a special case of Theorem 2.

Corollary 1: If $W(B, S)$ computes F over a closed domain, then $F(x)$ is a loop constant.

Proof: Define the equivalence R by xRy iff $F(x) = F(y)$. Clearly, x_0, x_f belongs to the same equivalence class. Hence, applying Theorem 2, $F(x)$ is a loop constant. □

Corollary 2: Under the conditions of Theorem 2,

$$x_f R x_0 \quad \text{iff} \quad [B(x) \Rightarrow S(x)\, Rx]. \qquad \square$$

The following example shows that a relation may hold initially and at termination, although not following every iteration.

Example 2:

while $v \neq 1$ **do**
  If odd(v) **then** $v := v + 1$ **else** $v = v/2$ **endif**
enddo
$D = \{v \mid v \geqslant 1$ and v integer$\}$.

It is required to show that the value of v does not increase as a result of execution of the loop. Thus, for any arbitrary u, we want to show

$$\{v \leqslant u\}\ W(B, S)\ \{v \leqslant u\}.$$

However, $v \leqslant u$ is not a loop invariant. □

It is interesting to note that, for Example 2, subgoal induction [23] generates the following verification condition which does not hold.

$$[v \neq 1 \wedge \text{odd}(v) \wedge v_f \leqslant v + 1 \Rightarrow v_f \leqslant v]$$
$$\wedge\ [v \neq 1 \wedge \text{even}(v) \wedge v_f \leqslant v/2 \Rightarrow v_f \leqslant v].$$

Example 3: The following program claims to compute the greatest common divisor of two positive integers m, n.

begin
  integer t;
  while $m \neq n$ **do**
    if $m < n$ **then** $t := m; m := n; n := t$ **endif**;
    $m := m - n$
  enddo
end;

Let $D = \{\langle m, n\rangle \mid m, n$ integer; $m, n > 0\}$. It is easy to show that D is closed. Let $GCD(x, y)$ denote the greatest common divisor of positive x, y. Let $H\langle m, n\rangle = \langle GCD(m, n), GCD(m, n)\rangle$. We wish to show that the loop computes H. Define the relation R on D as follows.

$$\langle m, n\rangle R \langle m', n'\rangle \quad \text{iff} \quad H\langle m, n\rangle = H\langle m', n'\rangle.$$

Clearly, R is an equivalence relation. If $\langle m_0, n_0\rangle$ denotes initial values, $\langle m, n\rangle$ denotes values at any iteration and $\langle m_f, n_f\rangle$ denotes final values, then in order to show that $\langle m_f, n_f\rangle R \langle m_0, n_0\rangle$ (using Corollary 2), we have to show that

$$\{m' = m, n' = n, m \neq n, m < n\}\ t := m; m := n; n := t;$$
$$m := m - n\ \{\langle m, n\rangle R \langle m', n'\rangle\}$$

and

$$\{m' = m, n' = n, m \neq n, m \geqslant n\}\ m := m - n$$
$$\{\langle m, n\rangle R \langle m', n'\rangle\}.$$

Simplifying, we have to prove the following two facts.

$$m < n \Rightarrow GCD(m, n) = GCD(n - m, m)$$
$$m > n \Rightarrow GCD(m, n) = GCD(m - n, n). \qquad \square$$

V. Computations with Nondeterministic Loops

Dijkstra [5] has introduced a nondeterministic guarded command loop of the following form.

do
  $B_1 \rightarrow S_1$
$\square\ B_2 \rightarrow S_2$
  ⋮
$\square\ B_n \rightarrow S_n$
od

The B_i's are Boolean expressions called "guards" and the S_i's are "statement lists" consisting of a sequence of one or more statements of the traditional kind. The intuitive meaning of this construct is that an "iteration" consists of arbitrarily choosing one guarded command whose guard is true and then executing the associated statement list. In case no guard is true, loop execution is terminated.

The schema may have local variables (which could carry values from one iteration to next) in addition to local variables of S_i's. However, as we have argued following Lemma 1, the local variables of the schema can be considered local to each S_i without affecting computation. Hence, we make no further mention of these variables.

We present two theorems in connection with nondeterministic loops that are direct counterparts of Theorems 1 and 2. We first define closure for nondeterministic loops.

Definition: D is closed with respect to a given nondeterministic loop of the above type if

$$\{D \wedge B_i\}\ S_i\ \{D\}, \qquad 1 \leqslant i \leqslant n.$$

Let $NDW(B, S)$ denote the nondeterministic loop shown above.

Theorem 3: Let D be closed with respect to $NDW(B, S)$. For a variable z, $z_f = g(x_0)$ if and only if the following three conditions hold.

1) $NDW(B, S)$ terminates for every input x_0 from D.
2) $[D(x) \wedge_{i=1}^{n} \neg B_i(x)] = [g(x) = z]$.
3) $g(x)$ is a loop constant, i.e.,

$$[D(x) \wedge B_i(x)] \Rightarrow [g(x) = g(S_i(x))], \qquad 1 \leqslant i \leqslant n. \qquad \square$$

Theorem 4: Let D be closed with respect to $NDW(B, S)$. Let R be any equivalence relation on D. $x_0 R x_f$, for every initial value x_0 and final value x_f from D, if and only if $[B_i(x) \Rightarrow S_i(x)\, Rx], 1 \leqslant i \leqslant n$. □

It follows from these theorems that a nondeterministic loop must be carefully constructed if the goal is to compute a given function or an equivalence relation on termination. Every statement list S_i *must* preserve the function value or

the equivalence relation. Conversely, this requirement guarantees the computation of the desired result.

VI. Systematic Generation of Loop Invariants

There are two natural ways of looking at loop invariants. A loop invariant could be a proposition about "what has been done" or a proposition about "what remains to be done." In the first approach, the current values of global variables are related to the original values of those variables. Then it is shown that on termination, the desired function has been computed. The second approach, the one advocated in this paper, relates the output from current values of global variables to the output from initial values of these variables. Both approaches are illustrated in the following example.

Example 4: This program sums an array $A[1 \cdots n]$ of real numbers.

```
{i = 0 and sum = 0 and n ≥ 0}
while i≠n do
   i := i+1; sum := sum+A[i]
enddo
```

One possible invariant (using the first approach) relates the (current) value of "sum" to the array elements.

$$\text{sum} = \sum_{k=1}^{i} A[k].$$

Coupled with the loop exit condition ($i = n$), it proves that variable sum holds the sum of array elements at termination. The second approach is based on Theorem 1, which says that the eventual result obtained by starting from the values of sum, A, i at an arbitrary iteration is the same as starting with their initial values. Such a loop invariant is

$$\text{sum} + \sum_{k=i+1}^{n} A[k] = \text{sum}_0 + \sum_{k=i_0+1}^{n} A[k]. \qquad \square$$

An observation of the second approach appears in London [17], who attributes it to Jim King. It is quite likely that there may not be any simple relationship among variables as required by the first approach. For instance, in Example 1, it is difficult to state w as a function of u, v, u_0, v_0. However, Theorem 1 guarantees that there is an invariant of the second kind provided the function is defined over a closed domain. The first approach seems to be more natural; hence, most people tend to think in terms of "variable values at a general iteration" when confronted with formulating a loop invariant for a new problem. The second approach is not as intuitive. In fact, when both approaches are applicable to a problem, the second approach leads to a somewhat more complicated form of the invariant. We have, however, found that the second approach is far superior in its generality, and with practice one can learn to use it quite effectively.

Theorem 1 provides a technique for generating a (suitable) loop invariant when the input domain D is closed. We next consider the problem when D is not closed. The most important case of a nonclosed domain arises when the loop is preceded by an initialization. Such a program can be converted to a loop program by removing the initialization and by suitably restricting the domain D to the initial values of variables. However, then D is nonclosed if any of the initialized variables receives a different value inside the loop.

In case D is nonclosed, the following technique may be used whereby Theorem 1 can be applied. Find D', a superset of D that is closed with respect to $W(B, S)$. Next, find a function F' on D' such that $W(B, S)$ computes F over D if and only if it computes F' over D'. Then Theorem 1 may be applied to prove/disprove that $W(B, S)$ computes F' over D', which in turn would prove/disprove that it computes F over D.

One rule for finding D' and F' is to somehow identify the values of global variables that arise in different iterations, starting from the set of input values. If a closed form expression describing these intermediate values can be found, then we have located a domain D' that is closed with respect to $W(B, S)$. Usually F' is quite easy to guess once we have such a D'.

However, it may sometimes be quite difficult to find a D' as described above. In that case, we may let D' be the set of all possible values, sometimes suitably specialized, such as all integers; then D' is trivially closed. F' is usually more difficult to find in such a case. In the next section, we illustrate the power of this approach by specifying the invariants for certain program classes.

A. Two Classes of Naturally Provable Programs

As we have noted earlier, initialization creates a nonclosed domain for which Theorem 1 is not directly applicable. We show that for certain program classes (having special properties), one may obtain generalizations of Theorem 1 that allow initialization. We investigate programs of the form

```
Initialization;
while B do S enddo;
```

Accumulating Loop Programs: For a large number of loops, the initialization is of a particularly simple kind that involves a single variable z initialized to z_0. Furthermore, the body of the loop only changes the value of z, but does not branch depending on the value of z. Several typical examples are shown below.

Example 5:

a) (exponentiation)

```
z := 1
while v≠0 do
   if odd (v) then z := z*u endif;
   v := v/2; u := u*u
enddo
```

b) Finding the sum of an array sequence $A[i]$ through $A[n]$.

```
z := 0;
while i ≤ n do
   z := z + A[i]; i := i+1
enddo
```

c) Removing blanks from a string. Input is a string s; output is string z. Null denotes the null string, || denotes the concatanation operator, head(s) is the first character of string

when $s \neq$ null, tail(s) is the string obtained from s after removing its first character when s is nonnull.

```
z := null;
while s≠null do
   if head(s) = " "then
      z := z || head(s) endif;
   s := tail(s)
enddo
```
□

It should be noted that in the above examples z *accumulates* the results. Typically, it is required to show at termination of the loop that z holds a certain function value of the input variable values. We show that the loop invariant can be generated (deterministically) for such problems under a few mild restrictions that seem to be almost always met in practice.

We require that the program meet the following conditions; in the following x denotes the values of global variables (other than z).

Condition 1): Closure with respect to the domain of x. This is the usual closure condition applicable to all variable values except that of z. Let D denote the closed domain.

Condition 2): z is an accumulating variable.

a) The values of x are independent of that of z. In other words, if the initial value of z is modified, x is not affected. This is guaranteed if z does not appear in the right-hand side of an assignment when some other variable is on the left-hand side. (A similar rule can be given when procedure calls are allowed.)

b) Neither z (nor any other variable whose value depends on z) governs a conditional branch inside the loop. Thus, z only accumulates the result, but does not guide the course of computation.

Condition 3): Assignments to z are of a simple type.

a) Every assignment to z inside the loop is of the form $z := z \oplus f(x)$ where $\oplus$ is an associative operator that is identical for every assignment; f is any function of global variables that may differ for different assignments.

b) Initial assignment is $z := z_0$ where z_0 is the unique left and right unity of $\oplus$.

We emphasize that these conditions seem to be met in practice in a large number of loop programs.

Theorem 5: Let T be any program satisfying conditions 1), 2), and 3) given above. Let x_0 denote the initial values of global variables (other than z) and z_f denote the final value of z. Then (assuming program termination), $z_f = g(x_0)$ for some function g if and only if,

a) $z \oplus g(x)$ is a loop constant (or $z \oplus g(x) = g(x_0)$ is a loop invariant)

b) $\neg B(x) \wedge x \in D \Rightarrow g(x) = z_0$. □

A proof of this result appears in [2], [21].

The first condition implies that

$$z_0 \oplus g(x_0) = \cdots = z \oplus g(x) = \cdots = z_f \oplus g(x_f).$$

This condition says that the value of the current accumulating variable z and the values of global variables x would lead to the same final value as z_0 and x_0. The crucial part (only if, of the theorem) is proven through an appeal to Theorem 1.

Example 5 (continued):

a) In order to show that $z_f = u_0^{v_0}$, it is necessary and sufficient to show (following proof of termination)

i) $z * u^v = u_0^{v_0}$ is a loop invariant and

ii) $v = 0 \Rightarrow (u^v = 1)$.

b) To show that $z = \sum_{k=i}^{n} A[k]$, we need to prove

i) $z + \sum_{k=i}^{n} A[k] = \sum_{k=i_0}^{n} A[k]$ is a loop invariant and

ii) $i > n \Rightarrow \left[\sum_{k=i}^{n} A[k] = 0\right]$.

c) To show that $z = NBL(s)$ where NBL is a suitably defined function, we need to show that

i) $z \,||\, NBL(s) = NBL(s_0)$ is a loop invariant and

ii) $[s = \text{null}] \Rightarrow [NBL(s) = \text{null}]$. □

A special case of Theorem 5 appears in [16].

Programming with a Stack: We consider the following schema where T is a stack.

```
T := (e0); {stack T is initialized to contain e0}
while T ≠ EMPTY do S enddo
```

Global variables have initial value x_0 and final value x_f. We do not have closure with respect to stack T, which is also a global variable to the loop. It is required to show that $x_f = x_0 \oplus e_0$ for some given binary operator $\oplus$. The special nature of stacks permits us to prove a theorem regarding the loop invariant. Let T be a stack consisting of elements $(e_1, e_2, \cdots, e_n)$ from top to bottom. Define

$$x \oplus' T = \begin{cases} x, & \text{if } T \text{ is empty} \\ ((\cdots((x \oplus e_1) \oplus e_2) \cdots) \oplus e_n), & \text{otherwise.} \end{cases}$$

We make the following restriction on the stack schema: the stack is not examined for emptiness in S. This guarantees that the stack elements will be processed in the intended manner from top to bottom. Without this restriction, a tricky programmer may rearrange the stack elements during an iteration by saving e_1, removing e_2, pushing e_1, pushing e_2, etc. We furthermore assume closure with respect to D, the domain of x.

Theorem 6: Let x denote any global variable values and let $[e]$ represent a stack containing a *single* element e. Let x' denote the global variable values and T' the stack content after *one iteration* through the loop.

$$x_f = x_0 \oplus e_0 \quad \text{if and only if,}$$
$$x \oplus e = x' \oplus T', \quad \text{for } \textit{every } x, e.$$ □

A proof of this theorem appears in [21].

Example 6 (Preorder Traversal of a Tree): The following program computes the sequence of nodes visited during preorder traversal of a binary tree using a stack T. The variable root is a pointer to the root of the tree to be traversed; the variable traversal denotes a sequence; || is the concatanation operator on a sequence; for a node P, left(P) and right(P)

are pointers to left and right sons (*nil* if there is no corresponding son), and name(P) denotes the name of the node to which P points.

```
traversal := null; {Initialize traversal to a null sequence}
T := (root); {Initialize T to contain the root node}
while T ≠ EMPTY do
begin
   P ⇐ T; {Pop the top off the stack}
   traversal := traversal || name(P);
   if right(P) ≠ nil then T ⇐ right(P) {Push onto stack}
      endif;
   if left(P) ≠ nil then T ⇐ left(P) endif
enddo
```

It is required to show that traversal_f = preorder (root) where preorder is a given function defined on the node of a tree that returns the sequence of nodes in preorder traversal. Preorder has to be defined formally in order to formally verify the above program. We can, however, state the conditions that need to be proven.

We first note that traversal behaves as an accumulating variable, || is an associative operator, and *null* is its (left and right) unity. Hence, we may consider the following input constraint, which ignores initialization to traversal.

Input Constraint: Root is a node of some tree; traversal is some sequence,

$$T = (\text{root}).$$

We will have to prove that with this input constraint

$$\text{traversal}_f = \text{traversal}_0 \,||\, \text{preorder (root)}.$$

This is in the form $x_f = x_0 \oplus e_0$.

We can define $\oplus'$ as follows:

$$\text{traversal} \oplus' T = \begin{cases} \text{traversal if } T = \text{EMPTY} \\ \text{traversal} \,||\, \text{preorder}(e_1) \,||\, \text{preorder}(e_2) \cdots ||\, \text{preorder}(e_n) \\ \text{if } T \text{ contains } (e_1, e_2, \cdots, e_n) \text{ from top to bottom.} \end{cases}$$

Theorem 6 can now be applied. We will have to show the following: starting with an *arbitrary* value of traversal and an arbitrary element p on the stack, if we get traversal$'$ and T' after one iteration, then

$$\text{traversal} \oplus p = \text{traversal}' \oplus' T'.$$

We may simplify this statement by considering the four different cases as follows. (Note that traversal$'$ = traversal || name(p). Furthermore, traversal $\oplus p$ = traversal || preorder(p).)

1) p has no son:

traversal || preorder(p) = traversal || name(p)

{stack is empty after the iteration}.

2) p has a left son but no right son:

traversal || preorder(p) = traversal || name(p) || preorder(left(p)) {stack contains left(p) after the iteration}.

3) p has a right son but no left son:

traversal || preorder(p)

= traversal || name(p) || preorder(right(p))

{stack contains right(p) after the iteration}.

4) p has both sons:

traversal || preorder(p)

= traversal || name(p) || preorder(left(p))

|| preorder(right(p)) {stack contains left(p), right (p) in this order from top to bottom}.

These verification conditions can be further simplified by noting that $a \,||\, b = a \,||\, c$ implies $b = c$. We can completely avoid traversal and obtain the following conditions, proofs of which depend only on preorder.

1) p has no son:

preorder(p) = name(p).

2) p has a left son but no right son:

preorder(p) = name(p) || preorder(left(p)).

3) p has a right son but no left son:

preorder(p) = name(p) || preorder(right(p)).

4) p has both sons:

preorder(p)

= name(p) || preorder(left(p)) || preorder(right(p)). □

Correctness conditions could be derived in this case without a deep understanding of the functioning of the program.

VII. Related Work

Study of functions as computed by programs was initiated by Mills [19]. He showed that for any loop, successive values obtained after any number of iterations can be represented by rooted directed trees in the input domain where an arc from one value to another represents that the latter is obtained from the former after one iteration. This idea has been used to obtain the major results of this paper.

Subgoal induction, recently proposed in [23], seems to be a powerful scheme for proving most programs that arise in practice. There is a great deal of similarity between subgoal induction and the work reported here. However, there are some major differences. Subgoal induction applies to arbitrary relations; we only consider functional and equivalence relations. Subgoal induction is not guaranteed to prove a correct program correct. In particular, it is easily seen that it fails to prove Example 2. Our emphasis has been on necessary and sufficient conditions. Some of the results obtained in [20] imply that it is not possible to obtain such conditions unless

we deal with equivalence relations (or some slight generalizations) of equivalence relations.

Generation of loop invariants using difference equations has been suggested by Elspas [9]. In this scheme, the values of variables at a general ith iteration are expressed as functions of variable values at $(i-1)$th iteration. These equations are then "solved" and i is eliminated to obtain the desired invariant. A similar method using counters has been proposed by Katz and Manna [16].

Aside from the difficulty of systematically generating invariants by this technique, the variable values at the ith iteration may be difficult to specify even though the overall behavior may be quite simple to state. For instance, Example 1 manipulates w in such a fashion that the value of w at the ith iteration is difficult to state as a function of u, v, and their initial values. This technique seems to be useful when the body S of the loop is a straight line program so that the variable values depend in a certain simple way on the iteration number.

Another technique [25] is to push the output assertion q backward through the loop once, twice, three times, etc. to obtain assertions q_1, q_2, q_3, etc., which are successive approximations to the loop invariant. A human could possibly isolate the general pattern from these approximations. A dual technique is to execute the program forward symbolically to obtain a few forward patterns from which a general pattern could be deduced.

A method of recursion induction used by Topor [24] replaces the loop by an equivalent recursive procedure. It is interesting to note that Topor's replacements are always done in a fashion that ensures closure. The recursive version is then proven, which, along with the initialization, is then shown to imply the desired assertion.

Dijkstra [5] has suggested a different approach to program verification. Starting from the program specification, he constructs programs in such a manner that reasoning about correctness is done during the construction process, making it unnecessary to verify programs afterwards. In synthesizing loops from specifications, he often has to weaken (generalize) the post conditions to come up with invariants.

One general theme that stands out in all the methods is the notion of "generalization" of the proposition to be proven. A loop invariant is necessarily a generalization of the proposition to be proven in that the former captures the relationship among variables in all iterations, whereas the latter relates the variable values at termination. This generalization may be done over iterations, leading to a proposition with the iteration number i as an explicit or implicit parameter, which roughly states what conditions hold at a general ith iteration. This paper attempts a generalization in another direction, namely, the general function F' computed if the global variables are drawn from some closed domain D', which is a generalization of D. Theorem 1 guarantees the existence of a "simple" loop invariant of this nature.

Recursive programs are often much easier to prove than iterative ones. This phenomenon can be explained in terms of closure. Every recursive program is written in such a fashion that the computed values (of parameters) are legal inputs to the program. Hence, it follows that the recursive programs must be explicitly defined to compute a certain function over a closed domain. Thus, the proposition to be proven is trivially obtained from the function definition itself, without explicit recognition of the domain. A special case of Theorem 5 appears in [16].

VIII. Conclusion

We have argued in this paper that closure is a fundamental notion in dealing with loops. Any attempt at finding a loop invariant must explicitly or implicitly convert the problem to one over a closed domain. Once we have found the function computed by the loop over a closed domain, we can trivially generate the necessary and sufficient conditions for proof of this fact. In some sense, Theorem 1 implies that a loop invariant can be stated independent of how the loop operates; instead it depends only on what the loop does eventually.

Our attempt at applying these results to programs with initialization has met with a fair amount of success [21]. Most actual programs seem to be simple enough so that a closed domain and a corresponding function are usually easy to find. In fact, we have isolated several classes of programs that are naturally provable in this sense.

We believe that our ultimate ability in writing correct programs would be based on isolating such easily provable program structures.

Acknowledgment

I am indebted to Prof. S. K. Basu, who helped in formulating and developing a number of ideas in this paper and coauthored a previous paper. Dr. H. D. Mills introduced the author to this subject area; he stressed the importance of a functional approach to verification. Prof. R. Yeh and D. Matuszek of the University of Texas have spent a considerable amount of time reading several drafts of this paper and commenting on them.

I am grateful to the reviewers for the detailed technical comments and suggestions for improving the style of the paper. I would also like to thank D. Davis for her editorial assistance.

References

[1] S. Basu and J. Misra, "Proving loop programs," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 76-86, Mar. 1975.

[2] —, "Some classes of naturally provable programs," in *Proc. 2nd Int. Conf. on Software Eng.*, San Francisco, CA, Oct. 1976.

[3] R. Boyer and J. S. Moore, "Proving theorems about LISP functions," *J. Ass. Comput. Mach.*, vol. 22, pp. 129-144, Jan. 1975.

[4] M. Caplain, "Finding invariant assertion for proving programs," in *Proc. Int. Conf. on Reliable Software*, Los Angeles, CA, Apr. 1975.

[5] E. W. Dijkstra, *A Discipline of Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1976.

[6] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming.* New York: Academic, 1972.

[7] L. P. Deutsch, "An interactive program verifier," Ph.D. dissertation, Dep. Comput. Sci., Univ. California, Berkeley, June 1973.

[8] B. Elspas, K. N. Levitt, and R. J. Waldinger, "An interactive system for the verification of computer programs," SRI, Menlo Park, CA, Research Rep., Sept. 1973.

[9] B. Elspas, "The semiautomatic generation of inductive assertions for proving program correctness," SRI, Menlo Park, CA, Research Rep., July 1974.

[10] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. on Applied Math.*, vol. 19, J. T. Schwartz, Ed., Amer. Math. Soc., Providence, RI, 1967, pp. 19-32.

[11] D. I. Good, R. L. London, and W. W. Beldsoe, "An interactive program verification system," presented at the Int. Conf. on Reliable Software, Los Angeles, CA, Apr. 1975.
[12] S. M. German, "A program verifier that generates inductive assertions," B.A. thesis, Harvard Univ., Cambridge, MA, May 1974.
[13] I. Greif and R. Waldinger, "A more mechanical heuristic approach to program verification," in *Proc. Int. Symp. on Programming*, Paris, France, Apr. 1974, pp. 83-90.
[14] C. A. R. Hoare, "An axiomatic basis of computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, pp. 576-580, 583, Oct. 1969.
[15] S. Igarashi, R. L. London, and D. C. Luckham, "Automatic program verification, AIM-200," Stanford Artificial Intelligence Project, Stanford Univ., Stanford, CA, 1972.
[16] S. M. Katz and Z. Manna, "Logical analysis of programs," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 188-206, Apr. 1976.
[17] R. L. London, "A view of program verification," in *Proc. Int. Conf. on Reliable Software*, Los Angeles, CA, Apr. 1975.
[18] Z. Manna, *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974.
[19] H. D. Mills, "The new math of computer programming," *Commun. Ass. Comput. Mach.*, vol. 18, pp. 43-48, Jan. 1975.
[20] J. Misra, "Prospects and limitations of automatic assertion generation for loop programs," *SIAM J. Comput.*, Dec. 1977.
[21] —, "Systematic verification of simple loops," unpublished manuscript.
[22] M. S. Moriconi, "Towards the interactive synthesis of assertions," Univ. Texas at Austin, Res. Rep., Oct. 1974.
[23] J. H. Morris and B. Wegbreit, "Subgoal induction," *Commun. Ass. Comput. Mach.*, vol. 20, Apr. 1977.
[24] R. W. Topor, "Interactive program verification using virtual programs," Ph.D. dissertation, Dep. Artificial Intelligence, Univ. Edinburgh, Edinburgh, Scotland, 1975.
[25] B. Wegbreit, "The synthesis of loop predicates," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 102-112, Feb. 1974.

Jayadev Misra (S'71-M'72), for a photograph and biography, see p. 69 of the January 1978 issue of this TRANSACTIONS.