Technical Report TR-921 August 1980
AFOSR-F49620-80-C-001

A COMPARATIVE ANALYSIS OF FUNCTIONAL CORRECTNESS*

Douglas D. Dunlop and Victor R. Basili

Department of Computer Science
University of Maryland
College Park, MD 20742

*Research supported in part by the Air Force Office of Scientific Research Grant AFOSR-F49620-80-C-001

ABSTRACT

The functional correctness technique is presented and explained. An implication of the underlying theory for the derivation of loop invariants is discussed. The functional verification conditions concerning program loops are shown to be a specialization of the commonly used inductive assertion verification conditions. The functional technique is compared and contrasted with subgoal induction. Finally, the difficulty of proving initialized loops is examined in light of the inductive assertion and functional correctness theories.

Acknowledgement - The authors would like to thank Dr. Harlan Mills, who was the source of motivation for our studying the functional approach to correctness, for his insights into the technique and his open discussions on the work reported here.

1. Introduction

The relationship between programs and the mathematical functions they compute has long been of interest to computer scientists [McCarthy, 1963; Strachey, 1966]. More recently, [Mills, 1972, 1975] has developed a model of functional correctness, i.e. a technique for verifying a program correct with respect to an abstract functional specification. This theory has been further developed by [Basu & Misra, 1975; Misra, 1978] and now appears as a viable alternative to the inductive assertion verification method due to [Floyd, 1967; Hoare, 1969].

In order to describe the functional correctness model, we consider a program P with variables v1, v2, ... , vn. These variables may be of any type and complexity (e.g. reals, structures, files, etc.) but we assume each vi takes on values from a set di. The set D = d1 x d2 x ... x dn is the data space for P; an element of D is a data state. A data state can be thought of as an assignment of values to program variables and is written <c1,c2,...,cn> where each vi has been assigned the value ci in di.

The effect of a program can be described by a function f:D->D which maps input data states to output data states. If P is a program, the function computed by P, written [P], is the set of ordered pairs {(X,Y) | if P begins execution in data state X, P will terminate in final state Y}. The domain of [P] is thus the set of data states for which P terminates.

If the specifications for a program P can be formulated as a data state to data state function f, the correctness of a program can be determined by comparing f with [P]. Specifically, we say that P computes f if and only if f ⊂ [P]. That is, if f(X) = Y for some data states X and Y, we require that [P](X) be defined and be equal to Y. Note that in order for P to compute f, no explicit requirement is made concerning the behavior of P on inputs outside the domain of f.

Example 1: Consider the simple program

```
P = while a > 0 do
      b := b * a;
      a := a - 1
    od.
```

The function computed by the program can be written as

[P]={(<a,b>,<0,b*a!>) | a>=0} U {(<a,b>,<a,b>) | a<0}.

Thus if a is greater than or equal to zero, the program maps a and b to 0 and b*a! respectively, otherwise the program performs the identity mapping. As a notational convenience, we often use conditional rules ard data state to data state "assignments" (called concurrent assignments) to express functions. In this notation we have

[P] = (a>=0 -> a,b := 0,b*a! |TRUE -> a,b := a,b).

Finally, if we are given f = (a>=0 -> a,b := 0,b*a!) as the

function to be computed, we may say that P computes f, since f is a subset of [P].

2. The Functional Correctness Technique

The functional correctness method relies heavily on a technique for verifying that a WHILE loop computes a given state to state function. We present this WHILE loop technique as a theorem and then describe the method for general programs.

Notation: The domain of a function f will be written as D(f). The notation f o g will be used to represent the composition of the functions g and f. We will use the shorthand B*Q for the WHILE loop while B do Q od. Finally, in several examples we will use the notation SUM(a,b,c,d) for the summation from a=b to c of d.

Definition: The loop B*Q is closed for a function f if and only if for all X in D(f), B(X) implies [Q](X) is in D(f). Intuitively, a loop is closed for f if the data state remains in D(f) as it executes for any input in D(f).

Theorem 1: If the loop B*Q is closed for a function f, then the loop computes f if and only if, for all X in D(f)
(2.1) the loop terminates when executed in initial state X,
(2.2) B(X) -> f(X) = f([Q](X)), and
(2.3) ~B(X) -> f(X) = X.

Proof: First, suppose (2.1), (2.2), and (2.3) hold. Let X[0] be any element of D(f). By condition (2.1) the loop must produce some output after a finite number of iterations. Let n represent this number of iterations, and let X[n] represent the output of the loop. Furthermore, let X[1], X[2], ... X[n-1] be the intermediate states generated by the loop, i.e. for all i satisfying 0 <= i < n, we have B(X[i]) & X[i+1] = [Q](X[i]) and also ~B(X[n]). Condition (2.2) shows f(X[0]) = f(X[1]) = ... = f(X[n]). Condition (2.3) indicates f(X[n]) = X[n]. Thus f(X[0]) = X[n] and the loop computes f.

Secondly, suppose the loop computes f. This fact would be contradicted if (2.1) were false. Suppose (2.2) were false, i.e. there exists an X in D(f) for which B(X) but f(X) <> f([Q](X)). From the closure requirement, [Q](X) is in D(f) and the loop produces f([Q](X)) when given the input [Q](X). But this implies the loop can distinguish between the cases where [Q](X) is an input and the case where [Q](X) is an intermediate result from the input X. However, this is impossible since the state describes the values of all program variables. Finally, if (2.3) were false, there would exist an X in D(f) for which the loop produces X as an output, but where f(X) <> X. Thus the loop must not compute f.

An important aspect of Theorem 1 is the absence of the need for an inductive assertion or loop invariant. Under the conditions of the theorem, a loop can be proven or disproven directly from its function specification.

Example 2: Using the loop P and function f of Example 1, we shall show P computes f. D(f) is the set of all states satisfying a >= 0. Since a is prevented from turning negative by the loop predicate, the loop is closed for f and Theorem 1 can be applied. The termination condition (2.1) is valid since a is decremented in the loop body and has a lower bound of zero. Since [Q](<a,b>) = <a-1, b*a>, condition (2.2) is

a > 0 -> f(<a,b>) = f(<a-1,b*a>)

which is

a > 0 -> <0,b*a!> = <0,b*a*(a-1)!>

which can be shown to be valid using the identity a! = a*(a-1)!. Condition (2.3) is

a = 0 -> <0,b*a!> = <a,b>

which is valid using the definition 0! = 1.

The functional correctness procedure is used to verify a program correct with respect to a function specification. Large programs must be broken down into subprograms whose intended functions may be more easily derived or verified. These results are then used to show the program as a whole computes its intended function. The exact procedure used to divide the program into subprograms is not specified in the functional correctness theory. In the interest of simplicity, the technique presented here is based on prime program decomposition [Linger, Mills & Witt, 1979]. That is, correctness rules will be associated with each prime program (or equivalently, with each statement type) in the source language. The reader should keep in mind, however, that in certain circumstances, other decomposition strategies may lead to more efficient proofs. One such circumstance is illustrated in Section 5.

In our presentation of the functional correctness procedure, we will consider simple Algol-like programs consisting of assignment, IF-THEN-ELSE, WHILE and compound statements. Before the correctness technique may be applied, the intended function of each loop in the program must be known. Furthermore, it is required that each loop be closed for its intended function. These intended functions must either be supplied with the program or some heuristic (not discussed here) must be employed by the verifier in order to derive a suitable intended function for each loop. This need for intended loop functions is analogous to the need for sufficiently strong loop invariants in an inductive assertion proof of correctness.

In order to prove that a structured statement S (i.e. a WHILE, IF-THEN-ELSE, or compound statement) computes a function f, it is necessary to first derive the function(s) computed by the component statement(s), and then to verify that S computes f using the derived subfunctions. Consequently, the function correctness technique will be described by a set of function derivation rules and a set of function verification rules:

Derive Rules - Used to compute [S].

D1: S = v:=e

1) Return [v:=e].

D2: S = s1;s2

1) Derive [S1]
2) Derive [S2]
3) Return [S2] o [S1].

D3: S = if B then S1 else S2 fi
1) Derive [S1]
2) Derive [S2]
3) Return (B->[S1] | TRUE->[S2]).

D4: S = while B do S1 od
1) Let f be the intended function (either given or derived)
2) Verify that while B do S1 od computes f
3) Return f.

Verify Rules - used to prove S computes f.

V1: S = v:=e
1) Derive [S]
2) Show f(X)=Y -> [S](X) = Y.

V2: S = S1;S2
1) Derive [S]
2) Show f(X)=Y -> [S](X) = Y.

V3: S = if B then S1 else S2 fi
1) Derive [S]
2) Show f(X)=Y -> [S](X) = Y.

V4: S = while B do S1 od
1) Derive [S1]
2) Apply Theorem 1.

Before considering an example of the use of these rules, we introduce two conventions that will simplify the proofs of larger programs. First, we allow an assignment into only a portion of the data state in a concurrent assignment. In this case it is understood that the other data state components are unmodified.

Example 3: If a program has variables v1,v2,v3, the sequence of assignments

v1 := 4; v3 := 7

performs the program function

v1,v3 := 4,7

which is shorthand for

v1,v2,v3 := 4,v2,7.

Secondly, if a function description is followed by a list of of variables surrounded by # characters, then the function is intended to describe the program's effect on these variables only. Other variables are considered to have been set to an undefined or unspecified value.

Example 4: If a program has variables v1,v2,v3 that take on values from d1,d2,d3, respectively, the function description

f = (v1 > 0 -> v2,v3 := v3,v2) #v2,v3#

is equivalent to

(v1 > 0 -> v1,v2,v3 := ?,v3,v2),

where ? represents an unspecified value. Note that in a sense,

functions like f are not data state to data state functions; more accurately they are general relations. E.g. in the example, <1,2,3> maps to <1,3,2> as well as <4,3,2>. However, we adopt the view that f is a d1 x d2 x d3 to d2 x d3 mapping and in this light, f is a functior. We call (v2,v3) the range set for f, written RS(f). Functions not using the # notation are assumed to have the entire set of variables as their range set. Similarly, if the variables vr1,vr2,...,vrk are the necessary inputs to a function description f, we say that (vr1,vr2,...,vrk) is the domain set for f, written DS(f). In Example 5, the domain set for f is (v1,v2,v3) which happens to be the entire set of variables, but this need not be the case. Note that some functions (e.g. constant functions) may have an empty domain set.

Note that the existence of functions with domain and range sets that are proper subsets of the entire set of variables has several implications for the Derive Rules given previously. In rule D2, we require that DS([S2]) C RS([S1]). If this is not the case, an intended function has been given with too small a range set. The resulting domain and range sets are given by

$$DS([S1;S2]) = DS([S1]) \cup DS([S2])$$
$$RS([S1;S2]) = RS([S2]).$$

In rule D3, the resulting domain and range sets are

$$DS([\underline{if}\ B\ \underline{then}\ S1\ \underline{else}\ S2\ \underline{fi}]) = DS([B]) \cup DS([S1]) \cup DS([S2])$$
$$RS([\underline{if}\ B\ \underline{then}\ S1\ \underline{else}\ S2\ \underline{fi}]) = RS([S1]) \cap RS([S2]).$$

Example 5: Consider the following program

```
S1)    (n>=0 -> s := SUM(i,1,m,i**n)) #s#
 1)       a := 1; s := 0;
S2)    (n>=1 -> s := s + SUM(i,a,m,i**n)) #s#
 2)       while a <= m do
 3)          i := 0; p := 1;
S3)        (n>=i -> p,i := p*a**(n-i),n)
 4)          while i<n do
 5)             i := i + 1;
 6)             p := p * a
 7)             od;
 8)          s := s + p;
 9)          a := a + 1
10)          od.
```

In this example, the functions labelled S1, S2 and S3 are the intended functions for the program, outer WHILE loop and inner WHILE loop respectively. We use the notation Fn-m as the derived function for lines n thru m of the program.

Step 1) - Using D1 and D2 we get
F5-6 = i,p := i+1,p*a.

Step 2) - We must verify the inner loop computes its intended function. The closure condition and termination condition are easily verified. The other conditions are

i<n -> <p*a**(n-i),n> = <p*a*a**(n-i-1),n>

and

$$i=n \rightarrow \langle p*a^{**}(n-i), n\rangle = \langle p, i\rangle$$

which are clearly true.

Step 3) - Using D1 and D2 we derive F3-7 as follows:

F3-7 = (n>=i -> p,i := p*a**(n-i),n) o F3-3
= (n>=i -> p,i := p*a**(n-i),n) o i,p := 0,1
= (n>=0 -> p,i := a**n,n).

Step 4) - Again with D1 and D2 we derive F3-9:

F3-9 = F8-9 o (n>=0 -> p,i := a**n,n)
= s,a := s*p,a+1 o (n>=0 -> p,i := a**n,n)
= (n>=0 -> p,i,s,a := a**n,n,s*a**n,a+1).

Step 5) - Now we are ready to show the outer loop computes its intended function. Again the closure and termination conditions are easily shown. The remaining conditions are

a<=m -> s*SUM(i,a,m,i**n) = s*a**n*SUM(i,a+1,m,i**n)

and

a>m -> s*SUM(i,a,m,i**n) = s,

both of which are true.

Step 6) - We now derive F1-10. Applying D2 we get

F1-10 = (n>=1 -> s := s * sum(i,a,m,i**n))#s# o f1-1
= (n>=1 -> s := s * sum(i,a,m,i**n))#s# o a,s := 1,0
= (n>=1 -> s := sum(i,1,m,i**n))#s#.

Step 7) - Since the intended program function agrees with F1-10, we conclude the program computes its intended function.

The functional correctness technique was developed by [Mills, 1972, 1975]. This verification method is compared and contrasted with the inductive assertion technique in [Basili & Noonan, 1978]. The presentation here emphasizes the distinction between function derivation and function verification in the correctness procedure.

In [Basu & Misra, 1975], the authors prove a result similar to Theorem 1 for the case where the loop contains local variables.

The closure requirement of Theorem 1 has recieved considerable attention. Several classes of loops which can be proved without the strict closure restriction are discussed in [Misra, 1978; Basu, 1980]. Results in [Wegbreit, 1977], however, indicate that, in general, the problem of "generalizing" a loop specification in order to satisfy the closure requirement is NP-complete.

3. The Loop Invariant f(X0) = f(X)

An important implication of Theorem 1 is that a loop which computes a function must maintain a particular property of the data state across iterations. Specifically, after each iteration, the function value of the current data state must be the same as the function value of the original input. In this section we discuss and expand on this characteristic of loops computing functions for which they are closed.

A loop assertion for the loop B*Q is a boolean-valued expression which yields the value TRUE just prior to each evaluation of the predicate B. In general, a loop assertion I is a function of the current values of the program variables (which we will denote by X), as well as the initial values of the program variables (denoted by X0). To emphasize these dependencies we write I(X0,X) to represent the loop assertion I.

Let D be a set of data states. A loop invariant for B*Q over a set D is a boolean valued expression I(X0,X) which satisfies the following conditions for all X0,X in D

(3.1) $I(X0,X0)$

(3.2) $I(X0,X) \;\&\; B(X) \rightarrow I(X0,[Q](X)) \;\&\; [Q](X)$ in D.

Thus, if I(X0,X) is a loop invariant for B*Q over D, then I(X0,X) is a loop assertion under the assumption the loop begins execution in a data state in D. Furthermore, the validity of this fact can be demonstrated by an inductive argument based on the number of loop iterations.

Loop assertions are of interest because they can be used to establish theorems which are valid when (and if) the execution of the loop terminates. Specifically, any assertion which can be inferred from

(3.3) $I(X0,X) \;\&\; \sim B(X)$

will be valid immediately following the loop.

It should be clear that for any loop B*Q, there may be an arbitrary number of valid loop assertions. Indeed, the predicate TRUE is a trivial loop assertion for any WHILE loop. However, the stronger (more restrictive) the loop assertion, the more one can conclude from condition (3.3). For a given state to state function f, we say that I(X0,X) is an f-adequate loop assertion iff I(X0,X) is a loop assertion and I(X0,X) can be used in verifying that the loop computes the function f. More precisely, if f is a function, the condition for a loop assertion I(X0,X) being an f-adequate loop assertion is

(3.4) $I(X0,X) \;\&\; \sim B(X) \rightarrow X = f(X0)$

for all X0 in D(f). A loop invariant I(X0,X) over some set containing D(f) for which condition (3.4) holds is an f-adequate loop invariant.

Example 6: Let P denote the program

```
while not a in (0,1) do
    if a > 0 then
        a := a - 2
    else a := a + 2 fi
```

odd.

Consider the following predicates

I1(a0,a) iff TRUE
I2(a0,a) iff abs(a) <= abs(a0)
I3(a0,a) iff odd(a) = odd(a0)
I4(a0,a) iff odd(a) = odd(a0) & abs(a) <= abs(a0)
I5(a0,a) iff odd(a) = odd(a0) | (a=3 & a0=2)

where abs denotes an absolute value function, and odd returns 1 if its argument is odd and 0 otherwise. Each of the 5 predicates is a loop assertion. Let D be the set of all possible data states for P (i.e. D = {<a> | a is an integer}). Let f = {(<a>,<odd(a)>)}, and consider I3. Since a in {0,1} implies a=odd(a), we can infer a=odd(a0) from I3(a0,a) & a in {0,1}. Thus I3 is an f-adequate loop assertion. Similarly, I4 and I5 are f-adequate loop assertions, but neither I1 nor I2 is restrictive enough to be f-adequate. Predicates I3 and I4 are loop invariants over D; however, since I5 fails (3.2) it is not a loop invariant (a=3,a0=2 is a counter example).

Theorem 2: If B*Q is closed for f and B*Q computes f then f(X0) = f(X) is an f-adequate loop invariant over D(f), and furthermore, it is the weakest such loop invariant in the sense that if I(X0,X) is any f-adequate loop invariant over D(f), I(X0,X) -> f(X)=f(X0) for all X,X0 in D(f).

Proof: First we show that f(X)=f(X0) is a loop invariant over D(f). Condition (3.1) is f(X0)=f(X0). From Theorem 1, for all X in D(f),

B(X) -> f(X) = f([Q](X)).

Thus for all X,X0 in D(f),

B(X) & f(X0)=f(X) -> f(X0)=f(X)=f([Q](X)) -> f(X0)=f([Q](X)).

Adding the closure condition B(X) -> [Q](X) in D(f) yields condition (3.2). Thus f(X)=f(X0) is a loop invariant over D(f). Again from Theorem 1, for all X in D(f),

~B(X) -> f(X)=X.

Thus for all X0 in D(f),

f(X)=f(X0) & ~B(X) -> f(X)=f(X0) & f(X)=X -> f(X0) = X

which shows f(X)=f(X0) is f-adequate. Let I(X0,X) be any f-adequate loop invariant for B*Q over D(f), and let Z0,Z be elements of D(f) such that I(Z0,Z). Since B*Q computes f and Z is in D(f), there exists some sequence Z[1],Z[2], ... ,Z[n] (possibly with n=1) where Z[1]=Z, Z[n]=f(Z), with B(Z[i]) & Z[i+1] = [Q](Z[i]) for all i satisfying 1 <= i < n. By condition (3.2) we have I(Z0,Z[1]), I(Z0,Z[2]), ... ,I(Z0,Z[N]); thus I(Z0,f(Z)) and ~B(f(Z)). Since Z0 is in D(f) and I(X0,X) is f-adequate,

I(Z0,f(Z)) & ~B(f(Z)) -> f(Z0)=f(Z)

from condition (3.4). Thus for all Z0,Z in D(f),

I(Z0,Z) -> f(Z0) = f(Z).

Example 6 (continued): In this example, I3 is of the form f(X)=f(X0). I3 is clearly weaker than the other f-adequate loop invariant I4. It is worth noting that I5 is weaker than I3, but I5 is not a loop invariant, and I2 is weaker than I3, but I2 is

not f-adequate. This situation is illustrated in Figure 1. The set of pairs {(a0,a)} is partitioned into 2 sets with a not in {0,1} on the left and a in {0,1} on the right. Note that I4 (or any other f-adequate loop invariant for that matter) is a subset of I3. Furthermore, each f-adequate loop assertion is identical where a is in {0,1}. This shaded region is precisely the set f.

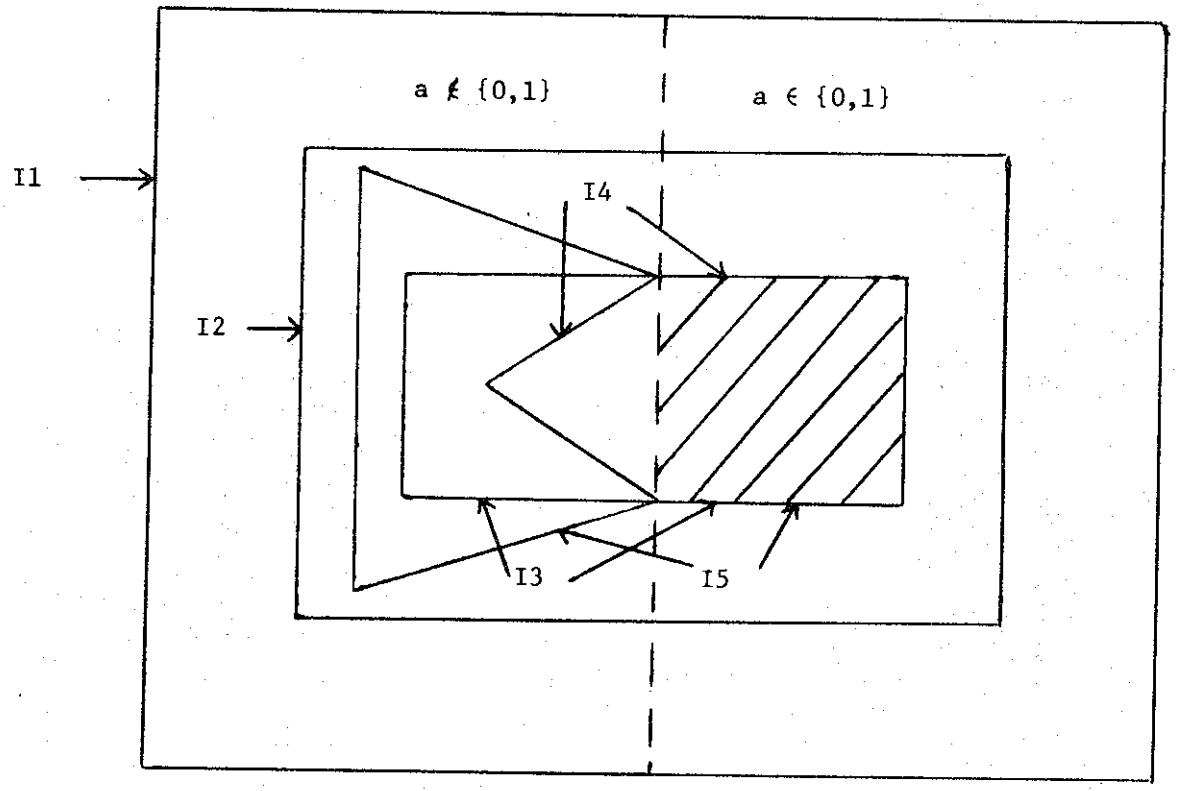

Figure 1.

Consider the problem of using Hoare's iteration axiom

(3.5) P & B {Q} P -> P {B*Q} P & ~B

to prove the loop B*Q computes a function f for which it is closed. In our terminology, P must be a loop invariant over some set containing D(f) (otherwise X=f(X0) for all X0 in D(f) cannot be inferred). However, using a loop invariant over a proper superset of D(f) is in general unnecessary, unless one is trying to show the loop computes some proper superset of f. If we choose to use a loop invariant P over exactly D(f), Theorem 2 tells us that f(X)=f(X0) is the weakest invariant that will do the job. In a sense, the weaker an invariant is, the easier it

is to verify that it is indeed a loop invariant (i.e. that the antecedant to (3.5) is true), because it says less (is less restrictive, is satisified by more data states, etc.) than other loop invariants. Along these lines, one might conclude that if a loop is closed for a function f, Theorem 2 gives a formula for the "easiest" loop invariant over D(f) that can be used to verify the loop computes f.

Let us again consider loop invariants and functions as sets of ordered pairs of data states. Let B*Q compute f and let I(X0,X) be an f-adequate loop invariant. We have seen that in this case

$$\{(X0,X) \mid I(X0,X) \ \&\ \neg B(X) \ \&\ X0 \text{ in } D(f)\}$$

is precisely f. That is, f must be the portion of the set represented by I(X0,X) obtained by restricting the domain to D(f) and discarding members whose second component cause B to evaluate to TRUE. Can the set represented by I(X0,X) be determined from f? No, since in general, there are many f-adequate invariants over D(f) and the validity of some will depend on the details of B and Q (e.g. I4 in Example 6). However, Theorem 2 gives us a technique for contructing the only f-adequate invariant over D(f) that will be valid for <u>any</u> B and Q, provided B*Q computes f and is closed for f. Specifically, this invariant couples each element of D(f) with its level set in f. Put another way - all f-adequate loop invariants over D(f) describe <u>what</u> the loop does (i.e. they can be used to show the loop computes f), and some may also contain information about <u>how</u> the final result is achieved. That is, one might be able to use an f-adequate loop invariant to make a statement about the intermediate states generated by the loop on some inputs. The intermediate states "predicted" by the weakest invariant f(X)=f(X0) is the set of all intermediate states that could possibly be generated by any loop B*Q that computes the function correctly. Thus, the invariant f(X)=f(X0) can be thought of as occupying a unique position in the spectrum of all possible loop invariants: it is strong enough to describe the net effect of the loop on the input set D(f) and yet is sufficiently weak that it offers no hint about the method used to achieve the effect.

<u>Example 7</u>: Consider the following program

```
while a > 0 do
    a := a - 1;
    c := c + b
od.
```

This loop computes the function

f = (a>=0 -> a,b,c := 0,b,c+a*b).

From Theorem 2, we know that

I(<a0,b0,c0>,<a,b,c>) iff <0,b0,c0+a0*b0>=<0,b,c+a*b>

is the weakest f-adequate invariant over D(f)={<a,b,c> | a>=0). Consider the sample input <4,10,7>. Our loop will produce the series of states <4,10,7>, <3,10,17>, <2,10,27>, <1,10,37>, <0,10,47>. Of course, our invariant agrees with these intermediate states (i.e. I(<4,10,7>,<4,10,7>), I(<4,10,7>,<3,10,17>), ... , I(<4,10,7>,<0,10,47>)), but it also

agrees with <6,10,-13>. We conclude then, that it is possible for some loop which computes f to produce an intermediate state <6,10,-13> while mapping <4,10,7> to <0,10,47>. Furthermore, no loop which computes f could produce <6,10,-12> as an intermediate state from the input <4,10,7> since the invariant would be violated.

To emphasize this point, we define an f-adequate invariant I(X0,X) over D(f) for B*Q to be an internal invariant if I(X0,X) implies that B*Q will generate X as an intermediate state when mapping X0 to f(X0). Intuitively, an internal invariant captures what the loop does as well as a great deal of how the loop works. In our example, b=b0 & c=c0+b*(a0-a) & 0<=a<=a0 is an internal invariant, but I(<a0,b0,c0>,<a,b,c>) as defined above is not (the state <6,10,-13> on input <4,10,7> is a counter example). It should be clear that if f has an infinite domain, no loop exists for which f(X)=f(X0) is an internal invariant. However, if we consider non-deterministic loops and weaken the definition of an internal invariant to one where I(X0,X) implies X may be generated by B*Q when mapping X0 to f(X0), such a loop can always be found. This loop would non-deterministically switch states so as to remain in the same level set of f. Our example program could be modified in such a manner as follows:

```
while a > 0 do
    t := "some integer value greater than or equal
                                to zero";
    c := c + b * (a-t);
    a := t
od
```

and corresponds to a "blind search" implementation of the function.

In [Basu & Misra, 1975], the authors emphasize the difference between locp invariants and loop assertions. The fact that f(X) = f(X0) is an f-adequate loop invariant appears in [Basu & Misra, 1975; Linger, Mills & Witt, 1979]. The independence of this loop invariant from the characteristics of the loop body is discussed in [Basu & Misra, 1975].

4. Comparison of the Hoare and Mills Loop Verification Rules

An alternative to using Theorem 1 in showing a loop computes a function is to apply Hoare's axiomatic verification technique. That is, one could verify P {B*Q} R where

P iff X=X0 in D(f), and
R iff X=f(X0)

by demonstrating the following for some predicate I:

(A1) P-> I
(A2) B & I {Q} I
(A3) ~B & I -> R.

Strictly speaking, conditions A1 thru A3 show partial correctness; to show total correctness, one must also prove

(A4) B*Q terminates for any input state satisfying P.

Note that if B*Q is closed for f, a predicate I that satisfies A1 and A2 is a loop invariant over D(f) (or some superset thereof).

We now wish to compare these verification conditions with the functional verification conditions. Recalling from Theorem 1, if B*Q is closed for f, the functional verification rules are:

(F1) B*Q terminates for any input state in D(f)
(F2) B(X) -> f(X) = f([Q](X)) for all X in D(f)
(F3) ~B(X) -> f(X) = X for all X in D(f).

In the following discussion we adopt the convention that if f is a function and X is not in D(f), then f(X)=Z is false for any Z.

Theorem 3: Let B*Q be closed for f. If f(X)=f(X0) is used as the loop invariant I in A1-A3, then A1 & A2 & A3 & A4 iff F1 & F2 & F3. That is, the functional verification conditions F1-F3 are equivalent to the special case of the axiomatic verification conditions A1-A4 which results from using f(X)=f(X0) as the loop invariant I. In particular, if I iff f(X)=f(X0) in the axiomatic rules, then

A1 is true,
A2 iff F2 provided X in D(f) & B(X) -> X in D([Q]),
A3 iff F3,
A4 iff F1.

Proof: We begin by noting that the termination conditions A4 and F1 are identical, thus A4 iff F1. Secondly A1 is

X=X0 in D(f) -> f(X)=f(X0)

which is clearly true for any f. Combining with our first result yields A1 & A4 iff F1. Condition A3 can be rewritten as

~B(X) & f(X)=f(X0) -> X=f(X0)

which is trivially true for any X,X0 outside D(f). Thus A3 may be rewritten as

(A3') For all X,X0 in D(f), ~B(X) & f(X)=f(X0) -> X=f(X0).

Note that A3' -> F3 by considering the case where X=X0. Furthermore, by adding f(X)=f(X0) to the antecedant of F3 we get

F3 -> (~B(X) & f(X)=f(X0) -> f(X)=X & f(X)=f(X0) -> f(X0)=X),

thus F3 -> A3'. Now we have A3 iff A3' iff F3 and adding this to our result above we get A1 & A3 & A4 iff F1 & F3. We next prove A2 & A4 iff F2 & F1. This combined with the above equivalence

yields the desired result A1 & A2 & A3 & A4 iff F1 & F2 & F3. Note that if there exists and X in D(f) such that B(X) but [Q](X) is not defined, then the loop itself will be undefined for X, both A4 and F1 will be false and A2 & A4 iff F2 & F1. We now consider the other case where for all X in D(f), B(X) -> X in D([Q]). In this situation we will show A2 iff F2; combining with A4 iff F1 yields A2 & A4 iff F2 & F1. Rule A2 may be rewritten as

$$B(X) \;\&\; f(X) = f(X0)\ \{Q\}\ f(X) = f(X0)$$

which again is trivially true if X or X0 is outside D(f); thus A2 is equivalent to

For all X,X0 in D(f), B(X) & f(X)=f(X0) {Q} f(X)=f(X0).

Since Q terminates for any input X in D(f) such that B(X) by hypothisis, this may be transformed to

(A2′) For all X,X0 in D(f), B(X) & f(X)=f(X0) -> f([Q](X))=f(X0).

As before, we can show A2′->F2 by considering the case where X=X0, and F2->A2′ by adding f(X)=f(X0) to the antecedant of F2. Thus A2 iff A2′ iff F2 which implies A2 iff F2. This completes the proof.

The purpose of Theorem 3 is to allow us to view the functional verification conditions as verification conditions in an inductive assertion proof. Not surprisingly, both techniques have identical termination requirements. If the termination condition is met, F2 amounts to a proof that f(X)=f(X0) is a loop invariant. Condition F3 amounts to a "Rule of Consequence", testing that the desired result can be implied from the loop invariant f(X)=f(X0) and the negation of the predicate B.

5. Subgoal Induction and Functional Correctness

Subgoal induction is a verification technique due to [Morris & Wegbreit, 1977]. In this section we compare subgoal induction to Mills' functional correctness approach.

We first note that subgoal induction can be viewed as a generalization of the functional approach presented here in that subgoal induction can be used to prove a program correct with respect to a general input-output relation. A consequence of this generality, however, is that the subgoal induction verification conditions are sufficient but not necessary for correctness; that is, in general, no conclusion can be drawn if the subgoal induction verification conditions are invalid. Provided the closure requirement is satisfied, the functional verification conditions (as well as the subgoal induction verification conditions when applied to functional specifications) are sufficient and necessary conditions for correctness. Results in [Misra, 1977] suggest that it is not possible to obtain necessary verification conditions for general input-output relations.

In order to more precisely compare the two techniques, we consider the flow chart program in Figure 2 taken from [Morris & Wegbreit, 1977].

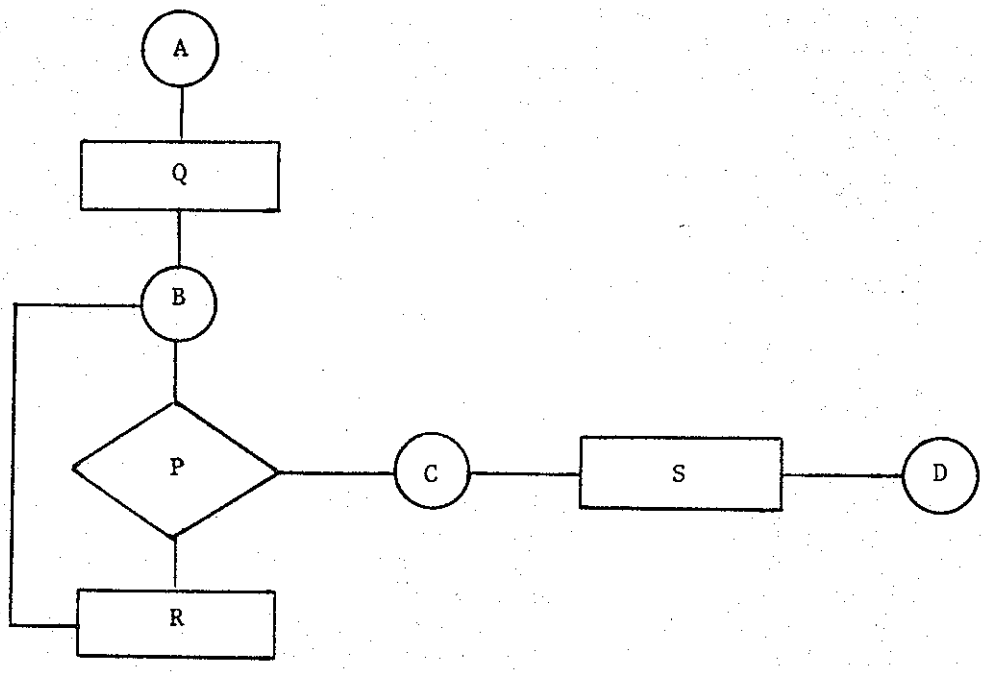

Figure 2.

In the figure, A,B,C,D are points of control in the flow chart, P is a predicate and Q,R and S are function nodes. Note that this flow chart program amounts to a WHILE loop surrounded by pre and post processing. Our goal is to prove the program computes a function T. Morris & Wegbreit point out that subgoal induction uses an induction on the B to D path of the flow chart; that is, one selects some relation V, inductively shows it holds for all B to D paths, and then uses V to show T is computed by all A to D paths. In our illustration, since T is a function, it will be required that V itself be a function. Once V has been selected, the verification conditions are

(S1) ~P(X) -> V(X) = S(X)
(S2) P(X) -> V(R(X)) = V(X)
(S3) T(X) = V(Q(X)).

Note that S1 and S2 test the validity of V; S3 checks that V can be used to show T.

The functional verification theory presented here is similar with the exception that the function S is not included in the induction path. We select some function f and show it holds for all B to C paths (i.e. we show the WHILE loop computes f) and then use f to show T is computed by all A to D paths. Once f has been selected, the verification conditions are

(F1) ~P(X) -> f(X)=X
(F2) P(X) -> f(R(X)) = f(X)
(F3) T(X) = S(f(Q(X))).

Note that both techniques require the invention of an intermediate hypothesis which must be verified in a "subproof." This hypothesis is then used to show the program computes T. The function S in the flow chart program is absorbed into the intermediate hypothesis in the subgoal induction case; it is separate from the intermediate hypothesis in the functional case. Indeed, the two intermediate hypotheses are related by

V = S o f.

If S is a null operation (identity function), the intermediate hypotheses and verification conditions of the two techniques are identical. A difference between the two techniques, however, can be seen by examining the case where Q is a null operation. If the loop is closed for T, subgoal induction enjoys an advantage since T can be used as the intermediate hypothesis. That is, the subgoal induction verification conditions are simply

(S1') ~P(X) -> S(X) = T(X)
(S2') P(X) -> T(R(X)) = T(X).

In the functional case, one must still derive an hypothesis for the loop function f. A heuristic which can be applied here is to restrict one's attention to functions which are subsets of S**-1 o T. However, it is worth emphasizing that this rule need not completely specify f since, in general, S**-1 o T is not a function relation. Once f has been selected, the verification conditions are

(F1') ~P(X) -> f(X)=X
(F2') P(X) -> f(R(X))=f(X)
(F3') T(X) = S(f(X)).

The difference between the two techniques in this case is

due to the prime program decomposition nature of the functional correctness algorithm described in Section 2. A more efficient proof is realized by treating the loop and the function S as a whole. Accordingly, correctness rules for this program form might be incorporated into the prime program functional correctness method described earlier. The validity of these rules can be demonstrated in a manner quite similar to the proof of Theorem 1.

Example 8: We wish to show the program

```
while not x in {0,1,2,3} do
   if x < 0 then x := x + 4
            else x := x - 4 fi
od;
if x > 1 then x := x - 2 fi
```

computes the function T={(<x>,<odd(x)>)}. The subgoal induction verification conditions are

x in {0,1,2,3} -> S(x) = odd(x), and
x ~in {0,1,2,3} -> odd(R(x)) = odd(x), where

S(x) = if x > 1 then x-2 else x, and
R(x) = if x < 0 then x+4 else x-4.

Both these conditions are straightforward. Now let us consider the prime program functional case. Suppose we are given (or may derive) the intended loop function

f = {(<x0>,<x>) | x in {0,1,2,3} & x mod 4 = x0 mod 4}.

We can verify that the loop computes f by demonstrating F1′ and F2′. Condition F3′ uses f to complete the proof.

The difficulty with splitting up the program in this example is that it requires the verifier to "dig out" unnecessary details concerning the effect of the loop. One need not determine explicitly the function computed by the loop in order to prove the program correct. The only important loop effect (as far as the correctness of the program is concerned) is x in {0,1,2,3} and odd(x) = odd(x0). In this example, treating the program as a whole appears superior since it only tests for the essential characteristics of the program components.

It is worth observing that an axiomatic proof of a program of this form could be accomplished by using the loop invariant T(X) = T(X0). The verification conditions in this case would be equivalent to the subgoal induction verification conditions. Note that, in general (as in our example), T(X) = T(X0) is too weak an invariant to be f-adequate for the intended loop function f.

6. Initialized Loops

The preceeding section indicates that it is occasionally advantageous to consider a program as a whole rather than to consider its prime prcgrams individually. In this section we attempt to apply the same philosophy to the initialized loop program form.

We will again consider the program in Figure 2 with the understanding that S is a null operation. We want to prove that the program computes a function T, i.e. that T holds for all A to C paths. We have seen that prime program functional correctness involves an induction on the B to C program path using an intermediate hypothesis f. An inductive assertion proof would involve an induction on the A to B path using some loop invariant I(X0,X). This invariant differs from those discussed previously in that it takes into account the initialization for the loop. In this section we discuss briefly the difficulty of synthesizing the intermediate hypotheses f and I.

In order for the program to compute T, we must have Q(X)=Q(Y) -> T(X)=T(Y). Consequently, the relation represented by T o (Q**-1) is a function and is a candidate for the intermediate hypothesis f. Unfortunately, the domain of this function is the image of D(T) through Q, and since the purpose of the initialization is often to provide a specific "starting point" for the loop, the loop will seldom be closed for this function. Thus the problem of finding an appropriate f can be thought of as one of generalizing T o (Q**-1).

Example 9: We want to show the program

```
s := 0; i := 0;
while i < n do
    i := i + 1;
    s := s + a[i]
od
```

computes s:=SUM(k,1,n,a[k]). If Q represents the function performed by the initialization, T o (Q**-1) is

(s=0,i=0 -> s:=SUM(k,1,n,a[k])).

Note that the loop is not closed for this function. To verify the program using the functional method, this function must be generalized to a function such as

f = s := s + SUM(k,i+1,n,a[k]).

We now consider the relative difficulties of synthesizing a function f for which the loop is closed (for a functional proof) and synthesizing an adequate loop invariant (for an inductive assertion proof). If we have a satisfactory f, an appropriate hypothesis for a loop invariant is I(X0,X) iff f(Q(X0))=f(X). We now try to go the other way. Suppose we have I(X0,X), can we derive from that a function f for which the loop is closed? We motivate the result as follows: we could obtain an equivalent program by modifying the initialization to (non-deterministically) map X0 to X if I(X0,X) is true. The

modified program still computes the same function; if the initialization maps X0 to anything other than Q(X0), the effect will simply be to save the loop some number of iterations. By the same argument that was used to show the loop must compute T o (Q**-1), the program must also compute T o (I(X0,X)**-1). Note that the loop is necessarily closed for this function; otherwise the invariant would be violated. We conclude then that the synthesis of a function for which the loop is closed and the synthesis of a suitable invariant are equivalent problems in the sense that a solution to one problem implies a solution to the other problem. The translation between loop invariants and intermediate hypotheses in a subgoal induction proof is discussed in [Morris & Wegbreit, 1977].

Example 2 (continued): An inductive assertion proof of our program might use the invariant s=SUM(k,1,i,a[k]) & i<=n. Note that this invariant is essentially equivalent to f(Q(X0))=f(X) (where f and Q are as defined previously). Using the technique outlined above, we may derive from the invariant

f′ = (s=SUM(k,1,i,a[k]) & i<=n -> s:=SUM(k,1,n,a[k])).

Observe that this is quite different from the original f, but that f′ is quite satisfactory for a functional proof of correctness. It may seem puzzling that f′(Q(X0))=f′(X) is the constant invariant TRUE and yet Theorem 2 states that such an invariant must be f′-adequate. This is not a contradiction, however, since

TRUE & i>=n -> s=SUM(k,1,n,a[k])

is valid for any state in D(f′). Similarly, a functional proof that the loop computes f′ is trivial with the exception of verifying that the closure requirement is satisfied. This is no coincidence: proving closure is equivalent to demonstrating the validity of the loop invariant.

7. Summary

Our purpose has been to explain the functional verification technique in light of other program correctness theories. The functional technique is based on Theorem 1 which provides a method for proving/disproving a loop correct with respect to a functional specification for which it is closed.

In Theorem 2, a loop invariant derived from a functional specification is shown to be the weakest invariant over the domain of the function which can be used to test the correctness of the loop. Theorem 3 indicates that the functional correctness technique for loops is actually the special case of the axiomatic method that results from using this particular loop invariant as an inductive assertion. The significance of this observation is that functional correctness can be viewed either as an alternative correctness procedure to the inductive assertion method or as a heuristic for deriving loop invariants.

The subgoal induction technique seems quite similar to the functional method; the two techniques often produce identical verification conditions. We have, however, observed an example where the subgoal induction method appears superior to functional correctness based on prime program decomposition. More work appears necessary in precisely characterizing these situations and determining if there are circumstances under which the functional method is more advantageous than subgoal induction.

We have examined the inductive assertion and functional methods for dealing with initialized loops. We have shown that the problems of finding a suitable loop invariant and finding a function for which the loop is closed are identical. The result indicates that for this class of programs the two methods are theoretically equivalent; that is, there is no theoretical justification for selecting one method over the other.

8. References

1. Basili, V. R. and Noonan, R. E. A Comparison of the Axiomatic and Functional Models of Structured Programming, IEEE Transactions on Software Engineering, (to appear September, 1980).

2. Basu, S. A Note on Synthesis of Inductive Assertions, IEEE Transactions on Software Engineering, SE-6 (January, 1980).

3. Basu, S. and Misra J. Proving Loop Programs, IEEE Transactions on Software Engineering, SE-1 (March, 1975).

4. Floyd, R. W. Assigning Meanings to Programs, Proceedings of a Symposia in Applied Mathematics, 19 (1967), pp. 19-32.

5. Hoare, C. A. R. An Axiomatic Basis for Computer Programming, CACM, 12 (October 1969), pp. 576-583.

6. Linger, R. C., Mills, H. and Witt, B. I. Structured Programming Theory and Practice, Addison-Wesley (1979).

7. McCarthy, J. A Basis for a Mathematical Theory of Computation. In: Brafford, P., and Hirschberg, D. (eds.): Computer Programming and Formal Systems. Amersterdam, North Holland (1963) pp. 33-70.

8. Mills, H. D. Mathematical Foundations for Structured Programming, IBM Federal Systems Division, FSC 72-6012 (1972).

9. Mills, H. D. The New Math of Computer Programming, CACM, 18 (January 1975).

10. Misra, J. Prospects and Limitations of Automatic Assertion Generation for Loop Programs, SIAM J. Comput., (December 1977).

11. Misra, J. Some Aspects of the Verification of Loop Computations, IEEE Transactions on Software Engineering, SE-4 (November 1978), pp. 478-486.

12. Morris, J. H. and Wegbreit, B. Subgoal Induction, CACM 20 (April 1977), pp. 209-222.

13. Strachey, C. Towards a Formal Semantics. In: Steel, T. B., Jr. (ed.): Formal Language Description Languages for Computer Programming. Proc. IFIP Working Conf. 1964, Amsterdam, North-Holland (1966) pp. 198-220.

14. Wegbreit, B. Complexity of Synthesizing Inductive Assertions, J. Ass. Comput. Mach., Vol. 24, pp. 504-512 (July, 1977).

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR-921	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* A COMPARATIVE ANALYSIS OF FUNCTIONAL CORRECTNESS		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER TR-921
7. AUTHOR(s) Douglas D. Dunlop and Victor R. Basili		8. CONTRACT OR GRANT NUMBER(s) AFOSR-F49620-80-C-001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park, Maryland 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Math. & Info. Sciences, AFOSR Bolling AFB Washington, D. C. 20332		12. REPORT DATE August 1980 13. NUMBER OF PAGES 23
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

program verification
functional correctness
axiomatic correctness
subtoal induction
loop invariants

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The functional correctness technique is presented and explained. An implication of the underlying theory for the derivation of loop invariants is discussed. The functional verification conditions concerning program loops are shown to be a specialization of the commonly used inductive assertion verification conditions. The functional technique is compared and contrasted with subgoal induction. Finally, the difficulty of proving initialized loops is examined in light of the inductive assertion and functional correctness theories.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*