A Comparative Analysis of Functional Correctness

DOUGLAS D. DUNLOP AND VICTOR R. BASILI

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

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

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*assertion checkers*; *correctness proofs*; *validation*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*invariants*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Functional correctness, inductive assertion correctness, subgoal inductions

INTRODUCTION

The relationship between programs and the mathematical functions they compute has long been of interest to computer scientists [McCa63, Stra64]. More recently, Mills [Mill72, Mill75] has developed a model of functional correctness, that is, a methodology for verifying that a program is correct with respect to an abstract specification function. This theory has been further developed by Basu and Misra [Basu75, Misr78] and now appears as a viable alternative to the inductive assertion verification method that is due to Floyd and Hoare [Floy67, Hoar69].

This paper presents a tutorial view of the functional correctness theory. This view is based on a set of structured programming control structures. An implication that this verification theory has for the derivation of loop invariants is discussed. The functional verification technique is contrasted and compared with the inductive assertion and subgoal induction techniques using a common notation and framework. In this analysis, the functional verification conditions concerning program loops are shown to be quite similar to the subgoal induction verification conditions and a specialization of the commonly used inductive assertion verification conditions. Finally, the difficulty of proving initialized loops is examined in the light of the functional and inductive assertion theories.

1. DEFINITION OF TERMS

In order to describe the functional correctness model, we consider a program P with variables $v_1, v_2, \ldots, v_n$. These variables may be of any type and complexity (e.g., reals, structures, files), but we assume each v_i takes on values from a set d_i. The set $D = d_1 \times d_2 \times \cdots \times d_n$ is the *data space* for P, and 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 $\langle c_1, c_2, \ldots, c_n \rangle$ where each v_i has been assigned the value c_i in d_i.

D. D. Dunlop's present address is Intermetics, Inc., 4733 Bethesda Ave., Suite 415, Bethesda, MD 20814.

CONTENTS

The effect of a program can be described by a function $f: D \rightarrow D$ 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 $\{(X, Y) \mid$ 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 X 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 is a subset of $[P]$. 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

$$[P] = \{(\langle a, b\rangle, \langle 0, b * (a!)\rangle) \mid a \geq 0\}$$
$$\cup \{(\langle a, b\rangle, \langle a, b\rangle) \mid a < 0\}.$$

Thus if a is greater than or equal to zero, the program maps a to 0 and b to $b * (a!)$; if a is less than zero, the program maps a to a and b to b (the identity mapping).

As a notational convenience, we often use *conditional rules* and data-state-to-data-state "assignments" (called *concurrent assignments*) to express functions. In this notation, the function $[P]$ just described would be written

$$[P] = (a \geq 0 \rightarrow a, b := 0, b * (a!) \mid$$
$$\text{TRUE} \rightarrow a, b := a, b).$$

That is, if a is greater than or equal to zero, the effect of the function is described by the first concurrent assignment; otherwise (i.e., if TRUE holds), the effect of the function is described by the second concurrent assignment. Finally, if we are given $f = (a \geq 0 \rightarrow a, b := 0, b * (a!))$ as the function to be computed, we may say that P computes f, since f is a subset of $[P]$.

We make use of the following notation. The domain of a function f is written as $D(f)$. The notation $f \circ g$ is used to represent the composition of the functions g and f; that is, if $h = f \circ g$, then $h(X) = f(g(X))$. If g is a function or binary relation, g^{-1} represents the binary relation which corresponds to the inverse of g. If A and B are Boolean-valued formulas, the notation $A \leftrightarrow B$ states that A is true if and only if B is true. Finally, we use the shorthand $B*H$ for the **while** loop program:

while $B(X)$ **do**
$X := H(X)$
od.

In this program, X represents the program data state, B is a total predicate on the data state, and H is a data-state-to-data-state function which represents the input/output effect of the loop body.

2. THE FUNCTIONAL CORRECTNESS TECHNIQUE

The functional correctness technique relies heavily on a procedure whose responsibility is to verify that a **while** loop computes a given state-to-state function. We present this **while** loop procedure as a theorem and then describe the technique for general programs. We first need the following definition.

Definition

The loop $B*H$ is closed for a set of data states S if and only if (iff)

$$X \in S \;\&\; B(X) \rightarrow H(X) \in S.$$

Intuitively, a loop is closed for S if the data state remains in S as it executes for any input in S.

Theorem 1

If the loop $B*H$ is closed for the domain of a function f, then the loop computes f iff, for all $X \in D(f)$

$$\textit{the loop terminates when executed in the initial state } X, \tag{2.1}$$

$$B(X) \rightarrow f(X) = f(H(X)), \tag{2.2}$$

$$\sim B(X) \rightarrow f(X) = X. \tag{2.3}$$

Proof. First, suppose (2.1)-(2.3) hold. Let X_0 be any element of $D(f)$. By condition (2.1), the loop must halt for the input X_0, thereby producing 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 $X_1, X_2, \ldots, X_{n-1}$ be the intermediate states generated by the loop, so that for all i satisfying $0 \leq i < n$, we have $B(X_i)$ and $X_{i+1} = H(X_i)$, and also $\sim B(X_n)$ holds. Condition (2.2) shows that $f(X_0) = f(X_1) = \ldots = f(X_n)$. Condition (2.3) indicates that $f(X_n) = X_n$. Thus $f(X_0) = X_n$ and the loop computes f.

Second, suppose the loop computes f. Let us now consider the consequences if each condition in the theorem were false. If (2.1) were false, our supposition would be contradicted. If (2.2) were false, that is, there existed an $X \in D(f)$ for which $B(X)$ held but $f(X) \neq f(H(X))$, then, from the closure requirement, $H(X)$ would be in $D(f)$ and, when given the input $H(X)$, the loop would produce $f(H(X))$ (by the supposition). Since the loop produces $f(X)$ when presented with the input X (again, by the supposition), this would imply that the loop could distinguish between the case in which $H(X)$ was an input and the case in which $H(X)$ was an intermediate result from the input X. It is impossible that the loop could so distinguish, since the data state describes the values of all program variables and this, in turn, precludes the possibility of the loop's reacting differently in these two cases. Finally, if (2.3) were false, there would exist an $X \in D(f)$ for which the loop would produce X as an output, but where $f(X) \neq X$. Because none of the three cases bear out our original supposition, the loop must not compute f when one of (2.1), (2.2) and (2.3) is false. □

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

Example 2

Using the loop P and function f of Example 1, we show that P computes f. $D(f)$ is the set of all states satisfying $a \geq 0$. Since a is prevented from becoming negative by the loop predicate, the loop is closed for $D(f)$ 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 $H(\langle a, b\rangle) = \langle a - 1, b * a\rangle$, condition (2.2) is

$$a > 0 \rightarrow f(\langle a, b\rangle) = f(\langle a - 1, b * a\rangle)$$

which is

$$a > 0 \rightarrow \langle 0, b * (a!)\rangle = \langle 0, (b * a) * ((a - 1)!)\rangle$$

which can be shown to be valid using the associativity of $*$ and the definition $a! = a * ((a - 1)!)$. Condition (2.3) is

$$a = 0 \rightarrow \langle 0, b * (a!)\rangle = \langle a, b\rangle$$

which is valid using the definition $0! = 1$.

The functional correctness procedure is used to verify that a program is 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 can then be used to show that 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 tech-

nique presented here is based on prime program (i.e., language statement type) decomposition [LING79]. That is, correctness rules are 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 consider simple ALGOL-like programs consisting of assignment, **if-then-else** and **while** statements. Before we can apply the correctness technique, we must know the intended function of each loop in the program, and must be sure that each loop is closed for the domain of its intended function. These intended functions must be supplied either by the programmer or by us (in the role of the verifier). In the latter case, some heuristic (not discussed here) must be employed 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** statement, **if-then-else** statement, or sequence consisting of two statements) computes a function f, it is first necessary to *derive* the function computed by each component statement, and then to *verify* that S computes f using these derived subfunctions. Consequently, the functional correctness technique is described by a set of function derivation rules and a set of function verification rules. These rules are given in Figure 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 extend the concurrent assignment notation described above by allowing an assignment into only a portion of the data state. In this case it is understood that the other data-state components are unmodified. Before going on to the second convention, let us consider an illustration of this first one.

Example 3

If a program has variables v_1, v_2, v_3, the sequence of assignments

$$v_1 := 4;\ v_3 := 7$$

computes the function represented by the concurrent assignment

$$v_1, v_2, v_3 := 4, v_2, 7.$$

Our convention allows this function to be described using the shorthand notation

$$v_1, v_3 := 4, 7.$$

In this case we are assigning into the portion of the data state containing v_1 and v_3. The remaining portion (the variable v_2) is assumed to be unmodified.

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

Example 4

If a program has variables v_1, v_2, v_3 that take on values from d_1, d_2, d_3, respectively, the function description

$$f = (v_1 > 0 \rightarrow v_2, v_3 := v_1 + v_3, v_2)\ \#v_2, v_3\#$$

is equivalent to

$$(v_1 > 0 \rightarrow v_1, v_2, v_3 := ?, v_1 + v_3, v_2),$$

where ? represents any arbitrary value.

Note that in a sense, functions like f of Example 4 are not data-state-to-data-state functions but are more accurately considered general relations. For instance, if v_1 is greater than zero, this function f maps the input $\langle v_1, v_2, v_3\rangle$ to $\langle 1, v_1 + v_3, v_2\rangle$ as well as to $\langle 2, v_1 + v_3, v_2\rangle$. However, we adopt the view that the function f of Example 4 maps the set $d_1 \times d_2 \times d_3$ to the set $d_2 \times d_3$ and in this light, f is a function. We call $\{v_2, v_3\}$ the *range set* for f, and notate it RS(f). Functions not using the # notation are assumed to have the entire set of variables as their range set. Similarly, if the variables $vr_1, vr_2, \ldots, vr_k$ are the necessary inputs to a function description f, we say that $\{vr_1, vr_2, \ldots, vr_k\}$ is the *domain set* for f, and notate it DS(f). In Example 4, the

Rule	Statement Type	Steps	Comments
D1	$S = v := e$	1) Return $[v := e]$.	The function computed the assignment
D2	$S = S_1; S_2$	1) Derive $[S_1]$ 2) Derive $[S_2]$ 3) Return $[S_2] \circ [S_1]$	Use derivation rules Use derivation rules Compose the functions
D3	$S =$ **if** B **then** S_1 **else** S_2 **fi**	1) Derive $[S_1]$ 2) Derive $[S_2]$ 3) Return $(B \rightarrow [S_1] \mid TRUE \rightarrow [S_2])$	Use derivation rules Use derivation rules Conditional function
D4	$S =$ **while** B **do** S_1 **od**	1) Let f be the intended function (either given or derived) 2) Verify that **while** B **do** S_1 **od** computes f 3) Return f	 Use verification rules

(a)

Rule	Statement Type	Steps	Comments
V1	$S = v := e$	1) Derive $[S]$ 2) Show $f(X) = Y \rightarrow [S](X) = Y$	Use derivation rules Prove the implication
V2	$S = S_1; S_2$	1) Derive $[S]$ 2) Show $f(X) = Y \rightarrow [S](X) = Y$	Use derivation rules Prove the implication
V3	$S =$ **if** B **then** S_1 **else** S_2 **fi**	1) Derive $[S]$ 2) Show $f(X) = Y \rightarrow [S](X) = Y$	Use derivation rules Prove the implication
V4	$S =$ **while** B **do** S_1 **od**	1) Derive $[S_1]$ 2) Apply Theorem 1	Use derivation rules

(b)

Figure 1. (a) Derivation rules used to compute $[S]$; (b) verification rules used to prove that S computes f.

domain set for f is $\{v_1, v_2, v_3\}$, which happens to be the entire set of variables, but this need not be the case. Some functions (i.e., constant functions) may have an empty domain set.

Example 5

Consider the following program.

```
C1)  (n ≥ 0 → s := SUM(i, 1, m, i^n)) #s#
 1)    a := 1; s := 0;
C2)  (n ≥ 0 → s := s + SUM(i, a, m, i^n)) #s#'
 2)    while a ≤ m do
 3)        j := 0; p := 1;
C3)      (n ≥ j → p, j := p * a^(n-j), n)
 4)        while j < n do
 5)          j := j + 1;
 6)          p := p * a
 7)          od;
 8)        s := s + p;
 9)        a := a + 1
10)        od.
```

In this example, the functions on the lines labeled C1, C2 and C3 are program comments and define the intended functions for the program, the outer **while** loop and the inner **while** loop, respectively. If a is a variable and b, c and d are expressions, we use the notation SUM(a, b, c, d) for the quantity

$$\sum_{a=b}^{c} d.$$

Furthermore, we use the notation F_{n-m} to represent the derived function for lines n through m of the program.

Step 1. Using derive rules D1 and D2 we get

$$F_{5\text{-}6} = j, p := j + 1, p * a.$$

Step 2. We must verify that the inner loop computes its intended function. The

closure condition and termination condition are easily verified. The other conditions,

$$j < n \rightarrow \langle p * a^{(n-j)}, n\rangle = \langle (p * a) * a^{(n-(j+1))}, n\rangle$$

and

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

are true by the definitions $x^{(y+1)} = x * (x^y)$ and $x^0 = 1$.

Step 3. Using D1 and D2 we derive $F_{3\text{-}7}$ as follows:

$$\begin{aligned} F_{3\text{-}7} &= (n \geq j \rightarrow p, j := p * a^{(n-j)}, n) \circ F_{3\text{-}3} \\ &= (n \geq j \rightarrow p, j := p * a^{(n-j)}, n) \\ &\quad \circ j, p := 0, 1 \\ &= (n \geq 0 \rightarrow p, j := a^n, n). \end{aligned}$$

Step 4. Again with D1 and D2 we derive $F_{3\text{-}9}$:

$$\begin{aligned} F_{3\text{-}9} &= F_{8\text{-}9} \circ (n \geq 0 \rightarrow p, j := a^n, n) \\ &= s, a := s + p, a + 1 \\ &\quad \circ (n \geq 0 \rightarrow p, j := a^n, n) \\ &= (n \geq 0 \rightarrow p, j, s, a \\ &\quad := a^n, n, s + a^n, a + 1). \end{aligned}$$

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 (where $n \geq 0$)

$$a \leq m \rightarrow s + \mathrm{SUM}(i, a, m, i^n) = (s + a^n) + \mathrm{SUM}(i, a + 1, m, i^n)$$

and

$$a > m \rightarrow s + \mathrm{SUM}(i, a, m, i^n) = s,$$

both of which are true by virtue of the properties of the function SUM.

Step 6. We now derive $F_{1\text{-}10}$. Applying D2 we get

$$\begin{aligned} F_{1\text{-}10} &= (n \geq 0 \rightarrow s := s \\ &\quad + \mathrm{SUM}(i, a, m, i^n)) \; \#s\# \circ F_{1\text{-}1} \\ &= (n \geq 0 \rightarrow s := s \\ &\quad + \mathrm{SUM}(i, a, m, i^n)) \; \#s\# \\ &\quad \circ a, s := 1, 0 \\ &= (n \geq 0 \rightarrow s := \mathrm{SUM}(i, 1, m, i^n)) \\ &\quad \#s\#. \end{aligned}$$

Step 7. Since the intended program function (given in line C1) agrees with $F_{1\text{-}10}$, we conclude that the program is correct with respect to its specification.

The essential ideas behind the functional correctness technique just illustrated were developed by Mills [MILL72, MILL75]. Our presentation here is based on prime program decomposition of composite programs and emphasizes the distinction between function derivation and function verification in the correctness procedure. Basili and Noonan [BASI80] have compared and contrasted the functional correctness and inductive assertion verification theories.

The essential idea behind Theorem 1 can be traced to McCarthy [MCCA62, MCCA63], who described a technique, called "recursion induction," for proving two functions equivalent. Theorem 1 can be viewed as a specific application of McCarthy's technique. Manna and Pnueli [MANN70, MANN71], and more recently, Topor [TOPO75] and Morris and Wegbreit [MORR77], have suggested loop verification rules similar to that stated in Theorem 1. Basu and Misra [BASU75] have proved a result corresponding to Theorem 1 for the case in which the loop contains local variables.

The closure requirement of Theorem 1 has received considerable attention. Several classes of loops which can be proved without the strict closure restriction have been analyzed by Misra and Basu [BASU76, MISR78, MISR79, BASU80]. Wegbreit [WEGB77], however, has described a class of programs for which 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 that computes a function must maintain a particular property of its 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 requirement, specifically focusing on loops which are closed for the domain of the functions they compute.

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

Let D be a set of data states. A *loop invariant* for $B*H$ over a set D is a Boolean-valued expression $A(X0, X)$ which satisfies the following conditions for all $X0, X \in D$:

$$A(X0, X0) \tag{3.1}$$

$$A(X0, X) \,\&\, B(X) \rightarrow A(X0, H(X)) \,\&\, (H(X) \in D). \tag{3.2}$$

Thus, if $A(X0, X)$ is a loop invariant for $B*H$ over D, then $A(X0, X)$ is a loop assertion under the assumption that the loop begins execution in a data state in D. Furthermore, the validity of this deduction 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 conditions which are valid when (and if) the execution of the loop terminates. Specifically, any assertion that can be inferred from

$$A(X0, X) \,\&\, \sim B(X) \tag{3.3}$$

will be valid immediately following the loop.

It should be clear that for any loop $B*H$, there may be an arbitrary number of valid loop assertions. For instance, the predicate TRUE is a trivial loop assertion for any **while** loop. However, the stronger (more restrictive) the loop assertion, the more that one can conclude from condition (3.3). For a given state-to-state function f, we say that $A(X0, X)$ is an *f-adequate loop assertion* iff $A(X0, X)$ is a loop assertion and $A(X0, X)$ 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 $A(X0, X)$ being an f-adequate loop assertion is

$$X0 \in D(f) \,\&\, A(X0, X) \,\&\, \sim B(X) \rightarrow X = f(X0). \tag{3.4}$$

A loop invariant $A(X0, X)$ 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 $a \notin \{0, 1\}$ **do**
  if $a > 0$ **then**
    $a := a - 2$
  else $a := a + 2$ **fi**
  od.

Consider the following predicates:

$$A_1(\langle a0\rangle, \langle a\rangle) \leftrightarrow \text{TRUE}$$

$$A_2(\langle a0\rangle, \langle a\rangle) \leftrightarrow \text{ABS}(a) \leq \text{ABS}(a0)$$

$$A_3(\langle a0\rangle, \langle a\rangle) \leftrightarrow \text{ODD}(a) = \text{ODD}(a0)$$

$$A_4(\langle a0\rangle, \langle a\rangle) \leftrightarrow \text{ODD}(a) = \text{ODD}(a0) \,\&\, \text{ABS}(a) \leq \text{ABS}(a0)$$

$$A_5(\langle a0\rangle, \langle a\rangle) \leftrightarrow \text{ODD}(a) = \text{ODD}(a0) \vee (a = 3 \,\&\, a0 = 2)$$

where ABS denotes an absolute value function, and ODD returns 1 if its argument is odd and 0 otherwise. Each of the five predicates is a loop assertion. Let D be the set of all possible data states for P (i.e., $D = \{\langle a\rangle \mid a \text{ is an integer}\}$). Let $f = \{(\langle a\rangle, \langle \text{ODD}(a)\rangle) \mid a \text{ is an integer}\}$, and consider A_3. Since $0 = \text{ODD}(0)$ and $1 = \text{ODD}(1)$, $a \in \{0, 1\}$ implies $a = \text{ODD}(a)$. Thus we can infer $a = \text{ODD}(a0)$ from $A_3(\langle a0\rangle, \langle a\rangle) \,\&\, a \in \{0, 1\}$. Hence A_3 is an f-adequate loop assertion. Similarly, A_4 and A_5 are f-

adequate loop assertions, but neither A_1 nor A_2 is restrictive enough to be f-adequate. Predicates A_3 and A_4 are loop invariants over D; however, since A_5 fails (3.2) (use $a = 3$, $a0 = 2$ to see this), it is not a loop invariant.

Theorem 2

If $B*H$ is closed for $D(f)$ and $B*H$ 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 it is implied by all other such loop invariants. That is, if $A(X0, X)$ is any f-adequate loop invariant over $D(f)$, $A(X0, X) \rightarrow f(X) = f(X0)$ 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 $X \in D(f)$,

$$B(X) \rightarrow f(X) = f(H(X)).$$

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

$$B(X) \,\&\, f(X0) = f(X) \rightarrow f(X) = f(H(X)) \,\&\, f(X0) = f(X),$$

that is,

$$B(X) \,\&\, f(X0) = f(X) \rightarrow f(X0) = f(H(X)).$$

Adding the closure condition $B(X) \rightarrow H(X) \in D(f)$ yields condition (3.2). Thus $f(X) = f(X0)$ is a loop invariant over $D(f)$. Again from Theorem 1, for all $X \in D(f)$,

$$\sim B(X) \rightarrow f(X) = X.$$

Thus for all $X0 \in D(f)$,

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

that is,

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

which shows $f(X) = f(X0)$ is f-adequate. Let $A(X0, X)$ be any f-adequate loop invariant for $B*H$ over $D(f)$, and let $Z0$, Z be elements of $D(f)$ such that $A(Z0, Z)$. Since $B*H$ computes f and $Z \in D(f)$, there exists some sequence $Z_1, Z_2, \ldots, Z_n$ (possibly with $n = 1$) where $Z_1 = Z$, $Z_n = f(Z)$, $\sim B(Z_n)$, and with $B(Z_i)$ & $Z_{i+1} = H(Z_i)$ for all i satisfying $1 \leq i < n$. By condition (3.2) we have $A(Z0, Z_1)$, $A(Z0, Z_2)$, $\ldots$, $A(Z0, Z_n)$; thus $A(Z0, f(Z))$ and $\sim B(f(Z))$. Since $Z0 \in D(f)$ and $A(X0, X)$ is f-adequate,

$$A(Z0, f(Z)) \,\&\, \sim B(f(Z)) \rightarrow f(Z0) = f(Z)$$

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

$$A(Z0, Z) \rightarrow f(Z0) = f(Z). \qquad \square$$

Example 6 (continued)

In this example, A_3 is of the form $f(X) = f(X0)$. A_3 is weaker than the other f-adequate loop invariant A_4 since A_4 implies A_3. It is worth noting that A_3 is not weaker than A_5, but A_5 is not a loop invariant, and that A_3 is not weaker than A_2, but A_2 is not f-adequate. Thus indeed, A_3 is the weakest f-adequate loop invariant for P. This situation is illustrated in Figure 2, in which the sets labeled S_1–S_5 are the sets of ordered pairs $(\langle a0\rangle, \langle a\rangle)$ satisfying A_1–A_5, respectively; that is,

$$S_i = \{(\langle a0\rangle, \langle a\rangle) \mid A_i(\langle a0\rangle, \langle a\rangle)\}$$

for $i = 1, 2, 3, 4$ and 5. The diagram is partitioned in half with $a \notin \{0, 1\}$ on the left and $a \in \{0, 1\}$ on the right. Note that S_4 (or the set corresponding to any f-adequate loop invariant for that matter) is a subset of S_3. Furthermore, the sets corresponding to each f-adequate loop assertion (i.e., S_3, S_4 and S_5) are identical where $a \in \{0, 1\}$. This region of the diagram is precisely the set f.

Consider the problem of using Hoare's iteration axiom

$$I \,\&\, B\{X := H(X)\}I \rightarrow I\{B*H\}I \,\&\, \sim B \qquad (3.5)$$

to prove that the loop $B*H$ computes a function f where $B*H$ is closed for $D(f)$. In our terminology, if $B*H$ is correct with respect to f, I must be a loop invariant over (at least) the set $D(f)$ (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 generally unnecessary, unless one is trying to show that the loop computes some proper superset of f. If we

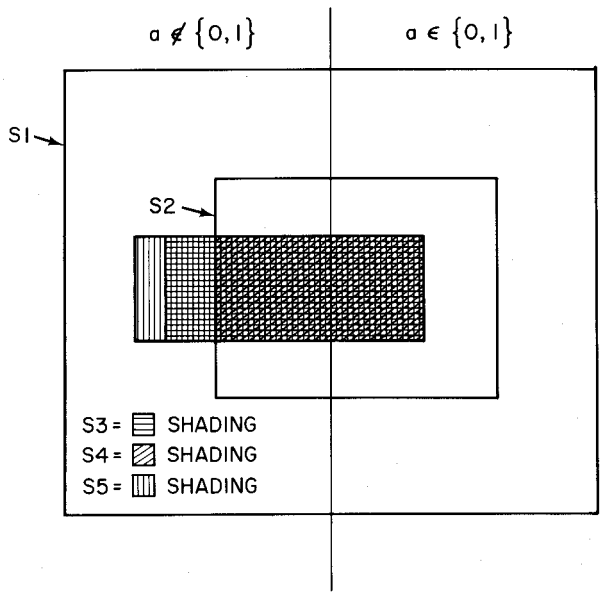

Figure 2. The sets S_1-S_5.

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

Let us again consider loop invariants and functions as sets of ordered pairs of data states. Let $B*H$ compute f and let $A(X0, X)$ be an f-adequate loop invariant. It is clear that in this case, the set

$$\{(X0, X) | A(X0, X) \ \& \sim B(X) \ \& \ (X0 \in D(f))\}$$

is precisely f. That is, f must be the portion of the set represented by $A(X0, X)$ obtained by restricting the domain to $D(f)$ and discarding members whose second component causes B to evaluate to TRUE (e.g., the portion of S_3 in Figure 2 in the region $a \in \{0, 1\}$ is f). Can the set represented by $A(X0, X)$ 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 H (e.g., A_4 in Example 6). However, Theorem 2 gives us a technique for constructing the only f-adequate invariant over $D(f)$ that will be valid for *any* B and H, provided $B * H$ computes f and is closed for $D(f)$. Specifically, this invariant couples an element of $D(f)$ with any other element of $D(f)$ which belongs to the same level set[1] of f.

Put another way, all f-adequate loop invariants over $D(f)$ describe *what* the loop does (i.e., they can be used to show that the loop computes f), and some may also contain information about *how* 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 $B*H$ that computes the function. 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 (i.e., specific enough) to describe the net effect of the loop on the input set $D(f)$ and

[1] S is a level set of f iff there exists a Y such that $S = \{X | f(X) = Y\}$.

yet is sufficiently weak (i.e., sufficiently general) that it offers no hint about the method used to achieve the effect.

Example 7

Consider the following program:

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

This loop computes the function

$$f = (a \geq 0 \rightarrow a, b, c := 0, b, c + a * b).$$

From Theorem 2, we know that

$$A(\langle a0, b0, c0\rangle, \langle a, b, c\rangle) \leftrightarrow \langle 0, b0, c0 + a0 * b0\rangle = \langle 0, b, c + a * b\rangle;$$

that is,

$$A(\langle a0, b0, c0\rangle, \langle a, b, c\rangle) \leftrightarrow b0 = b \;\&\; c0 + a0 * b0 = c + a * b$$

is the weakest f-adequate invariant over $D(f) = \{\langle a, b, c\rangle \mid a \geq 0\}$. Consider the sample input $\langle 4, 10, 7\rangle$. Our loop will produce the series of states $\langle 4, 10, 7\rangle$, $\langle 3, 10, 17\rangle$, $\langle 2, 10, 27\rangle$, $\langle 1, 10, 37\rangle$, $\langle 0, 10, 47\rangle$. Of course, our invariant agrees with these intermediate states (i.e., $A(\langle 4, 10, 7\rangle, \langle 4, 10, 7\rangle)$, $A(\langle 4, 10, 7\rangle, \langle 3, 10, 17\rangle)$, ..., $A(\langle 4, 10, 7\rangle, \langle 0, 10, 47\rangle)$), but it also agrees with $\langle 6, 10, -13\rangle$. We conclude then, that it is possible for some loop which computes f to produce an intermediate state $\langle 6, 10, -13\rangle$ while mapping $\langle 4, 10, 7\rangle$ to $\langle 0, 10, 47\rangle$. We further conclude that no loop which computes f could produce $\langle 6, 10, -12\rangle$ as an intermediate state from the input $\langle 4, 10, 7\rangle$, since the invariant would be violated.

To emphasize this point, we define an f-adequate invariant $A(X0, X)$ over $D(f)$ for $B*H$ to be an *internal invariant* if $A(X0, X)$ implies that $B*H$ 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,

$$(b = b0) \;\&\; (c = c0 + b * (a0 - a)) \;\&\; (0 \leq a) \;\&\; (a \leq a0)$$

is an internal invariant, but $A(\langle a0, b0, c0\rangle, \langle a, b, c\rangle)$ as defined above is not (since, for example, the program will not produce $\langle 6, 10, -13\rangle$ on the input $\langle 4, 10, 7\rangle$). It can be proved that if f is any nonempty function other than the identity function, no loop for computing f exists for which $f(X) = f(X0)$ is an internal invariant.[2] However, if we consider nondeterministic loops and weaken the definition of an internal invariant to be one where $A(X0, X)$ implies that X *may* be generated by $B*H$ when mapping $X0$ to $f(X0)$, such a loop can always be found. This loop would nondeterministically 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 would then correspond to a "blind search" implementation of the function.

Basu and Misra [BASU75] have emphasized the difference between loop invariants and loop assertions. The fact that $f(X) = f(X0)$ is an f-adequate loop invariant has been reported by several authors [BASU75, LING79]. The independence of this loop invariant from the characteristics of the loop body has been discussed by Basu and Misra [BASU75].

4. COMPARING THE HOARE AND MILLS LOOP PROOF RULES

An alternative to using Theorem 1, in showing that a loop computes a function, is to apply Hoare's inductive assertion verification technique [HOAR69]. In this technique, if R_1 and R_2 are predicates and T is a

[2] Outline of proof: Let $f(Y) \neq Y$ and $B*H$ compute f, and suppose that the f-adequate invariant $f(X) = f(X0)$ over $D(f)$ for $B*H$ is an internal invariant. We must have $B(Y)$. By (2.2) $f(Y) = f(H(Y))$. Consider $H(Y)$ as a fresh input. Since $f(X) = f(X0)$ is an internal invariant and $f(Y) = f(H(Y))$, the loop must eventually produce intermediate state Y, which must then produce $H(Y)$. Thus $B*H$ fails to terminate and does not compute f.

program, the notation R_1 $\{T\}$ R_2 asserts that if R_1 is true and the program T begins execution and terminates, then R_2 will be true. Applying Hoare's method, one could verify P $\{B*H\}$ Q where

$$P \leftrightarrow X = X0 \;\&\; X \in D(f),$$

$$Q \leftrightarrow X = f(X0)$$

by demonstrating the following for some predicate I:

A1: $P \rightarrow I$,
A2: $B \;\&\; I\{X := H(X)\}I$,
A3: $\sim B \;\&\; I \rightarrow Q$.

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

A4: $B*H$ terminates for any input state satisfying P.

We now wish to compare these verification conditions with the functional verification conditions. Recall from Theorem 1 that if $B*H$ is closed for $D(f)$, the functional verification rules are

F1: $X \in D(f) \rightarrow B*H$ terminates for the input X,
F2: $X \in D(f) \;\&\; B(X) \rightarrow f(X) = f(H(X))$,
F3: $X \in D(f) \;\&\; \sim B(X) \rightarrow f(X) = X$.

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 formula Z.

Theorem 3

Let $B*H$ be closed for $D(f)$. If $f(X) = f(X0)$ is used as the predicate I in A1–A3, then A1 & A2 & A3 & A4 $\leftrightarrow$ F1 & F2 & F3. That is, the functional verification conditions F1–F3 are equivalent to the special case of the inductive assertion verification conditions A1–A4 which results from using $f(X) = f(X0)$ as the predicate I. In particular, if $I \leftrightarrow f(X) = f(X0)$ in the inductive assertion rules, then

A1 $\leftrightarrow$ TRUE,
A2 $\leftrightarrow$ F2 provided $(X \in D(f)) \;\&\; B(X) \rightarrow X \in D(H)$,
A3 $\leftrightarrow$ F3, and
A4 $\leftrightarrow$ F1.

Proof. We begin by noting that the termination conditions A4 and F1 are identical; thus A4 $\leftrightarrow$ F1. Second, A1 is

$$X = X0 \;\&\; X \in D(f) \rightarrow f(X) = f(X0)$$

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

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

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

A3′: $X, X0 \in D(f) \;\&\; \sim B(X) \;\&\; f(X) = f(X0) \rightarrow X = f(X0)$.

Note that A3′ $\rightarrow$ F3 by considering the case where $X = X0$. Furthermore, F3 $\rightarrow$ A3′ by considering the case where $X0 \in D(f)$ and $f(X) = f(X0)$. Now we have A3 $\leftrightarrow$ A3′ $\leftrightarrow$ F3; adding this to A1 & A4 $\leftrightarrow$ F1 obtained above, we get the new equivalence A1 & A3 & A4 $\leftrightarrow$ F1 & F3. We next prove that A2 & A4 $\leftrightarrow$ F1 & F2. This, combined with the new equivalence, yields the desired result: A1 & A2 & A3 & A4 $\leftrightarrow$ F1 & F2 & F3. Note that if there exists an $X \in D(f)$ such that $B(X)$ holds but $H(X)$ is not defined, then the loop itself will be undefined for X, both A4 and F1 will be false, and A2 & A4 $\leftrightarrow$ F1 & F2.

We now consider the other case, where for all $X \in D(f)$, $B(X) \rightarrow X \in D(H)$. In this situation we will show that A2 $\leftrightarrow$ F2; combining this with A4 $\leftrightarrow$ F1 yields A2 & A4 $\leftrightarrow$ F1 & F2. Rule A2 may be rewritten as

$$(B(X) \;\&\; f(X) = f(X0))$$
$$\{X := H(X)\}\, f(X) = f(X0)$$

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

$$(X, X0 \in D(f) \;\&\; B(X) \;\&\; f(X) = f(X0))$$
$$\{X := H(X)\}\, f(X) = f(X0).$$

Since H is defined for any input $X \in D(f)$ such that $B(X)$ by hypothesis, this formula may be transformed using Hoare's axiom of assignment to the implication

A2′: $(X, X0 \in D(f) \;\&\; B(X) \;\&\; f(X) = f(X0)) \rightarrow f(H(X)) = f(X0)$.

As before, we can show A2′ $\rightarrow$ F2 by con-

sidering the case where $X = X0$, and F2 → A2′ by considering the case where $X0 \in D(f)$ and $f(X) = f(X0)$. Thus A2 ↔ A2′ ↔ F2 which implies A2 ↔ F2. This completes the proof of the theorem. □

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 predicate. Condition F3 amounts to an application of the "Rule of Consequence," ensuring that, given the predicate $f(X) = f(X0)$ and the negation of the predicate B, the desired result can be implied.

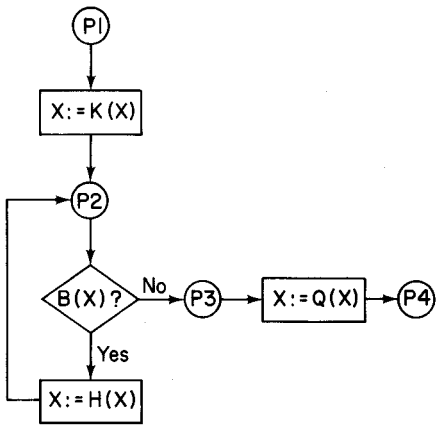

Figure 3. Flowchart program.

5. SUBGOAL INDUCTION AND FUNCTIONAL CORRECTNESS

Subgoal induction is a verification technique proposed by Morris and Wegbreit [MORR77]. It is based largely on work of Manna and Pnueli [MANN70, MANN71]. In this section we compare subgoal induction to the functional correctness approach described in Section 2.

We first note that subgoal induction can be viewed as a generalization of the functional approach 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 that the closure requirement is satisfied, the functional verification conditions (as well as the subgoal induction verification conditions, when applied to the same problem) are sufficient and necessary conditions for correctness. Results reported by Misra [MISR77] suggest that it is not possible to obtain necessary verification conditions for general input–output relations without considering the details of the loop body.

In order to compare the two techniques more precisely, we consider the flowchart program in Figure 3 adapted from the Morris and Wegbreit paper [MORR77]. In the figure, P1, P2, P3 and P4 are points of control in the flowchart, X represents the program data state, B is a predicate on the data state, and K, H and Q are data-state-to-data-state functions. Note that this flowchart program amounts to a **while** loop surrounded by pre- and postprocessing. Our goal is to prove that the program computes a function f. Morris and Wegbreit point out that subgoal induction uses an induction on the P2-to-P4 path of the computation; that is, one selects some relation v, shows inductively that it holds for all P2-to-P4 execution paths, and then, finally, uses v to show that f is computed by all P1-to-P4 execution paths. In our application, since f is a function, it will be required that v itself be a function. Once v has been selected, the verification conditions are

S1: $X \in D(v)\ \&\ \sim B(X) \rightarrow v(X) = Q(X)$,
S2: $X \in D(v)\ \&\ B(X) \rightarrow v(H(X)) = v(X)$,
S3: $X \in D(f) \rightarrow f(X) = v(K(X))$.

Note that S1 and S2 test the validity of v, and S3 checks that v can be used to show the program computes f.

An application of the functional verification technique presented here results in a similar kind of analysis except that the function Q is not included in the induction path. We select some function g and show that it holds for all P2-to-P3 execution paths (i.e., we show that the **while** loop computes g), and then use g to show that f

is computed by all P1-to-P4 execution paths. Once g has been selected, the verification conditions are

F1: $X \in D(g)$ & $\sim B(X) \rightarrow g(X) = X$,
F2: $X \in D(g)$ & $B(X) \rightarrow g(H(X)) = g(X)$,
F3: $X \in D(f) \rightarrow f(X) = Q(g(K(X)))$.

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 that the program computes f. The function Q in the flowchart 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

$$v = Q \circ g.$$

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

S1′: $X \in D(f)$ & $\sim B(X) \rightarrow Q(X) = f(X)$,
S2′: $X \in D(f)$ & $B(X) \rightarrow f(H(X)) = f(X)$.

In the functional case, one must still derive a hypothesis for the loop function g. A heuristic is to restrict one's attention to those functions which are subsets of $Q^{-1} \circ f$. However, it is worth emphasizing that this heuristic need not completely specify g since, in general, $Q^{-1} \circ f$ is not a function. Once g has been selected, the verification conditions are

F1′: $X \in D(g)$ & $\sim B(X) \rightarrow g(X) = X$,
F2′: $X \in D(g)$ & $B(X) \rightarrow g(H(X)) = g(X)$,
F3′: $X \in D(f) \rightarrow f(X) = Q(g(X))$.

The difference between the two techniques in this case is due to the prime program decomposition strategy employed in the functional correctness algorithm given in Section 2. A more efficient proof is realized by treating the loop and the function Q 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 that the program

while $x \notin \{0, 1, 2, 3\}$ **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 $f = \{(\langle x \rangle, \langle \mathrm{ODD}(x) \rangle)\}$. The subgoal induction verification conditions are

$$x \in \{0, 1, 2, 3\} \rightarrow Q(x) = \mathrm{ODD}(x),$$

$$x \notin \{0, 1, 2, 3\} \rightarrow \mathrm{ODD}(H(x)) = \mathrm{ODD}(x),$$

where

$$Q(x) = \text{if } x > 1 \text{ then } x - 2 \text{ else } x,$$

$$H(x) = \text{if } x < 0 \text{ then } x + 4 \text{ else } x - 4.$$

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

$$g = \{(\langle x0 \rangle, \langle x \rangle) \mid x \in \{0, 1, 2, 3\} \ \& \ x \bmod 4 = x0 \bmod 4\}.$$

We can verify that the loop computes g by demonstrating F1′ and F2′. Condition F3′ uses g 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 uncover these details, that is, 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 $\mathrm{ODD}(x) = \mathrm{ODD}(x0)$. 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, provided the loop is closed for $D(f)$, an inductive asser-

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

6. INITIALIZED LOOPS

The preceding section indicates that it is occasionally advantageous to consider a program as a whole rather than to consider its prime programs individually. In this section we attempt to apply the same philosophy to the initialized loop program form and use the result as a basis from which to compare the functional and inductive assertion approaches to this particular verification problem.

We again consider the program given in Figure 3, with the added understanding that Q is a null operation (identity function). We want to prove that the program computes a function f, that is, that f holds for all P1-to-P3 execution paths. We have seen that prime program functional correctness involves an induction on the P2-to-P3 execution path using an intermediate hypothesis g. An inductive assertion proof would involve an induction on the P1-to-P2 execution path using some limited loop invariant $A(X0, X)$ [LING79]. A limited loop invariant differs from those discussed previously in that it takes into account the initialization preceding the loop. One of the objectives of this section is to discuss the relative difficulties of synthesizing the intermediate hypotheses g and A.

We now reason about whether there might be an efficient way to verify the program by treating it as a whole (i.e., instead of treating the initialization and the loop individually). In order for the program to compute f, it must be that $K(X) = K(Y) \rightarrow f(X) = f(Y)$. Consequently, the relation represented by $f \circ (K^{-1})$ is a function and a candidate for the intermediate hypothesis g. Indeed, the initialized loop program is correct with respect to f iff $g = f \circ (K^{-1})$ is a function and the **while** loop (by itself) is correct with respect to g. Unfortunately, the domain of this function is the image of the set $D(f)$ mapped through the initialization K, 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 the domain of this function. Thus the problem of finding an appropriate intermediate hypothesis g can be thought of as one of generalizing $f \circ (K^{-1})$.

Example 9

We want to show that the program

$s := 0; i := 0;$
while $i < n$ **do**
  $i := i + 1;$
  $s := s + a_i$
od

computes

$$f = s := \mathrm{SUM}(k, 1, n, a_k)\ \#s\#.$$

As before, $\mathrm{SUM}(k, 1, n, a_k)$ is a notation for $a_1 + a_2 + \cdots + a_n$. If K represents the function performed by the initialization, $f \circ (K^{-1})$ is

$$(s = 0, i = 0 \rightarrow s := \mathrm{SUM}(k, 1, n, a_k))\ \#s\#.$$

Note that the loop is not closed for the domain of this function (since values of s and i different from 0 occur as the loop executes). To verify the program by means of the functional method, this function must be generalized to a function such as

$$g = s := s + \mathrm{SUM}(k, i + 1, n, a_k)\ \#s\#.$$

We now consider the relative difficulties of synthesizing a suitable loop function g (for a functional proof) and synthesizing an adequate limited loop invariant (for an inductive assertion proof). If we have a satisfactory g for a functional proof of the program, the analysis in Section 3 indicates that the invariant $A(X0, X) \leftrightarrow g(X) = g(X0)$ over $D(g)$ can be used to show that the loop computes g; absorbing the initialization $X := K(X)$ into the invariant gives the result that the limited invariant $A(X0, X) \leftrightarrow g(X) = g(K(X0))$ can be used to prove that the initialized loop program computes $g \circ K = f$. We now try to go the other way. Suppose we have an appropriate limited loop invariant $A(X0, X)$ for an inductive assertion proof of the program. Can we derive from that an adequate loop function

g? Yes, and we motivate the result as follows: We could obtain an equivalent program by modifying the initialization to map $X0$ to X (nondeterministically) if $A(X0, X)$ holds. The modified program (assuming termination) must still compute the same function; if the initialization maps $X0$ to anything other than $K(X0)$, the effect will simply be to alter the number of iterations executed by the loop. By the same argument used to show that the loop, assuming correctness, must compute $f \circ (K^{-1})$, the loop must also compute $f \circ (A(X0, X)^{-1})$. That is, if $A(X0, X)$ holds for some $X0 \in D(f)$ and for some X, the loop must map X to $f(X0)$. Note that the loop is necessarily closed for the domain of this function; otherwise the invariant would be violated. The proper conclusion is that the synthesis of an adequate loop function and the synthesis of a suitable invariant are equivalent problems in the sense that a solution to either problem implies a solution to the other problem.

The translation between loop invariants and intermediate hypotheses in a subgoal induction proof has been discussed by Morris and Wegbreit [MORR77] and King [KING80].

Example 9 (continued)

An inductive assertion proof of our program might use the limited invariant $s = \text{SUM}(k, 1, i, a_k)$ & $0 \leq i \leq n$. Note that this invariant implies the invariant $g(K(X0)) = g(X)$ discussed above (where g and K are as defined in Example 9). Using the technique outlined above, we may derive from this invariant the loop function

$$g' = (s = \text{SUM}(k, 1, i, a_k), 0 \leq i \leq n \rightarrow s := \text{SUM}(k, 1, n, a_k)) \; \#s\#.$$

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

$$\text{TRUE}, i \geq n \rightarrow s = \text{SUM}(k, 1, n, a_k)$$

is valid for any state in $D(g')$. Similarly, a functional proof that the loop computes g' 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 when the loop is closed for the domain of the function.

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 inductive assertion method that results from using this particular loop invariant as an inductive assertion. The significance of this observation is that the functional correctness technique for loops can be viewed either as an alternative verification 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 when 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 an adequate loop function are essentially 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.

ACKNOWLEDGMENTS

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.

We are grateful to the editor for suggesting many improvements in the presentation of this paper. The comments and suggestions made by the referees are very much appreciated.

This work was supported in part by the Air Force Office of Scientific Research Contract AFOSR-F49620-80-C-001 to the University of Maryland.

REFERENCES

BASI80 BASILI, V. R., AND NOONAN, R. E. "A comparison of the axiomatic and functional models of structured programming," *IEEE Trans. Softw. Eng.* **SE-6** (Sept. 1980), 454–464.

BASU75 BASU, S., AND MISRA, J. "Proving loop programs," *IEEE Trans. Softw. Eng.* **SE-1** (March 1975), 76–86.

BASU76 BASU, S. K., AND MISRA, J. "Some classes of naturally provable programs," in *Proc. 2nd Int. Conf. on Software Engineering* (San Francisco, Oct. 13–15), IEEE, New York, 1976, pp. 400–406.

BASU80 BASU, S. "A note on synthesis of inductive assertions," *IEEE Trans. Softw. Eng.* **SE-6** (Jan. 1980), 32–39.

FLOY67 FLOYD, R. W. "Assigning meanings to programs," *Proc. Symp. Appl. Math.* **19** (1967), 19–32.

HOAR69 HOARE, C. A. R. "An axiomatic basis for computer programming," *Commun. ACM* **12,** 10 (Oct. 1969), 576–583.

KING80 KING, J. "Program correctness: On inductive assertion methods," *IEEE Trans. Softw. Eng.* **SE-6** (Sept. 1980), 465–479.

LING79 LINGER, R. C., MILLS, H., AND WITT, B. I. *Structured programming theory and practice*, Addison-Wesley, Reading, Mass., 1979.

MCCA62 MCCARTHY, J. Towards a mathematical science of computation," in C. M. Popplewell (Ed.), *Proc. IFIP Congress 62*, North-Holland, Amsterdam, 1963, pp. 21–28.

MCCA63 MCCARTHY, J. "A basis for a mathematical theory of computation," in P. Brafford and D. Hirschberg (Eds.), *Computer programming and formal systems*, North-Holland, Amsterdam, 1963, pp. 33–70.

MANN70 MANNA, Z., AND PNUELI, A. "Formalization of properties of functional programs," *J. ACM* **17,** 3 (July 1970), 555–569.

MANN71 MANNA, Z. "Mathematical theory of partial correctness," *J. Comput. Syst. Sci.* **5** (June 1971), 239–253.

MILL72 MILLS, H. D. "Mathematical foundations for structured programming," FSC 72-6012, IBM Federal Systems Division, Bethesda, Md., 1972.

MILL75 MILLS, H. D. "The new math of computer programming," *Commun. ACM* **18,** 1 (Jan. 1975), 43–48.

MISR77 MISRA, J. "Prospects and limitations of automatic assertion generation for loop programs," *SIAM J. Comput.* **6** (Dec. 1977), 718–729.

MISR78 MISRA, J. "Some aspects of the verification of loop computations," *IEEE Trans. Softw. Eng.* **SE-4** (Nov. 1978), 478–486.

MISR79 MISRA, J. "Systematic verification of simple loops," Tech. Rep. TR-97, Univ. of Texas, Austin, Tex., March 1979.

MORR77 MORRIS, J. H., JR., AND WEGBREIT, B. "Subgoal induction," *Commun. ACM* **20,** 4 (April 1977), 209–222.

STRA64 STRACHEY, C. "Towards a formal semantics," in T. B. Steel, Jr. (Ed.), *Formal language description languages for computer programming*, Proc. IFIP Working Conf. 1964, North-Holland, Amsterdam, 1966, pp. 198–220.

TOPO75 TOPOR, R. W. "Interactive program verification using virtual programs," Ph.D. dissertation, Dep. of Artificial Intelligence, Univ. of Edinburgh, Scotland, 1975.

WEGB77 WEGBREIT, B. "Complexity of synthesizing inductive assertions," *J. ACM* **24,** 3 (July 1977), 504–512.

Received August 1981; final revision accepted March 1982.