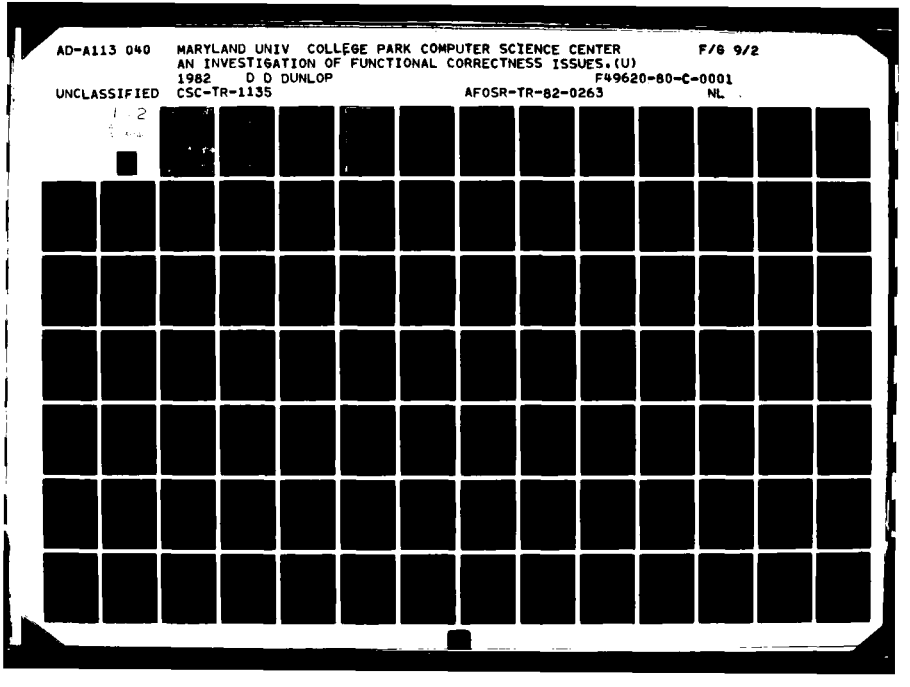
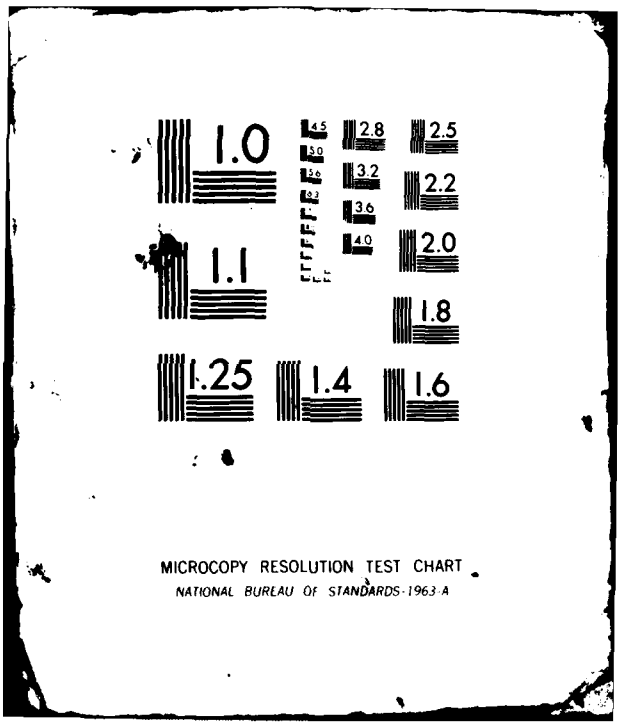
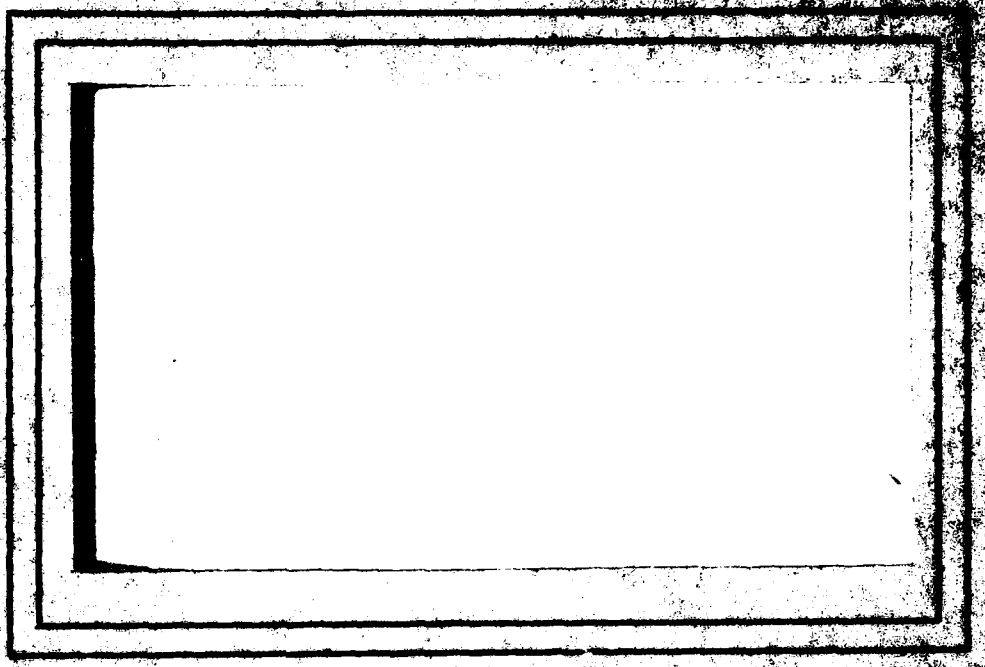

AD-A113 040
MARYLAND UNIV COLLEGE PARK COMPUTER SCIENCE CENTER F/G 9/2
AN INVESTIGATION OF FUNCTIONAL CORRECTNESS ISSUES.(U)
1982 D D DUNLOP F49620-80-C-0001
UNCLASSIFIED CSC-TR-1135 AFOSR-TR-82-0263 NL
1 2


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AFOSR-TR- 82-0208

AD A113040



DTIC
ELECTE
S MAY 4 1982 D

UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND
20742

DTIC FILE COPY

Technical Report TR-1135 January, 1982
[redacted]-F49620-80-C-0001

An Investigation of Functional Correctness Issues*

Douglas D. Dunlop

Dissertation submitted to the Faculty of the Graduate School of the University of Maryland in partial fulfillment of the requirements for the degree of Doctor of Philosophy
1982

---

*This work was supported in part by the Air Force Office of Scientific Research Contract [redacted]-F49620-80-C-0001 to the University of Maryland.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR- 82-0263 / 2. GOVT ACCESSION NO. AD-A113040	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): AN INVESTIGATION OF FUNCTIONAL CORRECTNESS ISSUES	5. TYPE OF REPORT & PERIOD COVERED: TECHNICAL
	6. PERFORMING ORG. REPORT NUMBER: TR-1135
7. AUTHOR(s): Douglas D. Dunlop	8. CONTRACT OR GRANT NUMBER(s): F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Center, University of Maryland, College Park MD 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Mathematical & Information Sciences Directorate, Air Force Office of Scientific Research, Bolling AFB DC 20332	12. REPORT DATE: JAN 82
	13. NUMBER OF PAGES: 95
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)

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

Given a program and an abstract functional specification that the program is intended to satisfy, a fundamental question is whether the program executes in accordance with (i.e. is correct with respect to) the specification. A simple functional correctness technique is initially defined which is based on prime program decomposition of composite programs. This technique is analyzed and the problems of the need for each intended loop function and the inflexibility of the prime program decomposition strategy are discussed. The (CONTINUED)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ITEM #20, CONTINUED: notion of a reduction hypothesis is then defined which can be used in the place of an intended loop function in the verification process. Furthermore, an efficient proof decomposition strategy for composite programs is suggested which is based on a sequence of proof transformations.

A heuristic technique is then proposed for synthesizing intended functions for WHILE loops in initialized loop programs. Although the technique seems to be useful in a wide range of commonly occuring applications, it is explained that the heuristic relies on the loop behaving in a 'reasonable' manner. A model of 'reasonable' loop behavior, a uniformly implemented loop, is defined. It is shown that, for any uniformly implemented loop, an inadequate intended loop function may be generalized (i.e. made more descriptive) in a systematic manner.

ABSTRACT

Title of Dissertation: AN INVESTIGATION OF FUNCTIONAL CORRECTNESS ISSUES

Douglas Dixon Dunlop, Doctor of Philosophy, 1982

Dissertation directed by: Dr. Victor R. Basili
Associate Professor
Department of Computer Science

Given a program and an abstract functional specification that the program is intended to satisfy, a fundamental question is whether the program executes in accordance with (i.e. is correct with respect to) the specification. A simple functional correctness technique is initially defined which is based on prime program decomposition of composite programs. This technique is analyzed and the problems of the need for each intended loop function and the inflexibility of the prime program decomposition strategy are discussed. The notion of a reduction hypothesis is then defined which can be used in the place of an intended loop function in the verification process. Furthermore, an efficient proof decomposition strategy for composite programs is suggested which is based on a sequence of proof transformations.

A heuristic technique is then proposed for synthesizing intended functions for WHILE loops in initialized loop programs. Although the technique seems to be useful in a wide range of commonly occurring applications, it is explained that the heuristic relies on the loop behaving in a "reasonable" manner. A model of "reasonable" loop behavior, a uniformly implemented loop, is defined. It is shown that, for any uniformly implemented loop, an inadequate intended loop function may be generalized (i.e. made more descriptive) in a systematic manner.

ACKNOWLEDGEMENTS

I would like to express my thanks to my advisor, Dr. Victor Basili, for the guidance and encouragement he has provided throughout my course of study at Maryland. Dr. Basili has put in many long hours making helpful comments on my work and proposing directions for research which often proved fruitful.

I would like to thank Dr. Harlan Mills for a number of beneficial discussions concerning this research effort. To a large extent, the work in this dissertation is inspired by Dr. Mills' research into the nature of loop computation. I would also like to thank Dr. Richard Hamlet, Dr. Mark Weiser, Dr. John Gannon, David Barton and Larry Morell for their helpful comments and suggestions on earlier drafts of the material in this dissertation.

Above all, I am grateful for the constant patience, support and encouragement provided by my wife, Janet Dunlop.

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.

Table of Contents

List of Figures

1. Introduction

The notion of program correctness is fundamental in software engineering and is of increasing importance as computers are used in more critical applications. Before being released for use, software usually undergoes a validation process, i.e. an act of reasoning in favor of its correctness. There exist two major validation approaches. The first is testing, and consists of executing the program on a number of inputs and then showing that the results are in agreement with the program specification. The alternative validation technique is program verification (or a proof of program correctness). A program is verified by applying, at some level of formality, logical and mathematical reasoning concerning the program and specification in order to certify their consistency. In this dissertation, we consider the verification approach to the program validation problem.

Unfortunately, a proof of correctness for any nontrivial program is a task that may be fraught with many difficulties. While our research addresses a number of these issues, there are others which we choose to ignore. For one thing, we assume that the program specification has been sufficiently formalized to make a rigorous proof of correctness possible. In practice, this can be a formidable task for programs which are intended to solve complex problems. Secondly, we will restrict our attention to procedureless, sequential, structured programs. We do not consider, therefore, the issues involved in proving programs containing global/local variables, GOTOs, parameter passing or concurrency.

Although our primary interest in studying verification is as a method of validating programs, research in this field often has implications for other areas of software engineering. For one thing, a crucial aspect of proving the correctness of a program is the act of reading and understanding its behavior. Any technique or methodology which facilitates this act is useful not only as a verification tool, but also as an aid in comprehending, documenting, modifying and maintaining programs. Secondly, verification research can provide insight into what constitutes a "good" program. Many practices associated with "structured programming," for example, are aimed at producing programs which can feasibly be proven correct. It is reasonable to expect that research into program characteristics which influence their verifiability should well have significant implications for the program development process.

Our approach to program verification is based largely on the work in [Mills 72, Mills 75]. This methodology has come to be known as functional correctness and centers around rules for proving a prime program (i.e. basic flow chart structure) correct with respect to a specification which has been formulated as a mathematical function. The goal of our research is a thorough study of a number of key issues dealing with the practical

application of the functional correctness methodology.

A straightforward version of functional correctness based on prime program decomposition of composite programs is defined in Chapter 2. This verification strategy is then compared and contrasted with other standard verification methodologies. In particular, the goal of the study is to analyze the theoretical relationships between functional, inductive assertion and subgoal induction proofs of correctness. In the analysis, it is observed that the prime program decomposition strategy causes some functional correctness proofs to be somewhat more complex than their inductive assertion or subgoal induction counterparts.

A primary impediment to applying functional correctness on a large scale in practice is the requirement that each WHILE loop in the program be tagged with an adequate intended function. Such a function describes the intended input/output behavior of the loop over a suitably general input domain. While supplying intended functions with program segments is no doubt a good programming practice, programmers often fail to supply this documentation due to the effort and difficulty involved. This is particularly true of intended loop functions (as opposed, for example, to those for initialized loops) since they tend to be relatively complex and difficult to state. Furthermore, most program maintenance is currently being done on programs with undocumented intended loop functions. Chapters 3, 4 and 5 deal largely with the issue of verifying programs which contain WHILE loops with missing or inadequate intended functions.

In Chapter 3, a new verification strategy for composite structured programs is described. Based on functional correctness, the technique is intended to be applied in the circumstance where there is no intended function supplied for a WHILE loop contained in the program. The method employs a reduction hypothesis which dispenses with the need for the intended loop function. Its merit is due to the fact that it may be easier to invent a reduction hypothesis than to create the intended loop function. Finally, the methodology uses the idea of a proof transformation to overcome the above mentioned deficiencies of prime program decomposition of composite programs.

In Chapter 4, we again consider the problem of a missing intended loop function but restrict our attention to initialized WHILE loop programs. The approach taken here is to create an adequate intended function for the WHILE loop and then to proceed with a standard functional correctness proof. A heuristic method is presented for synthesizing these functions. Although the heuristic seems useful in a wide range of cases, the point is made that its successful application relies on the program exhibiting a "reasonable" form of loop behavior.

The notion of a uniformly implemented loop is defined in Chapter 5. Our purpose here is twofold. First, the definition

is intended to characterize this idea of "reasonable" or "well structured" loop behavior. Secondly, the definition serves as the basis of a technique for systematically generalizing (i.e. making more descriptive) intended functions for uniformly implemented WHILE loops. The technique can be used to enhance a "skeleton" intended loop function to the point where it is adequate for a functional proof of correctness.

In Chapter 6, we conclude with a summary of our research and offer several suggestions for future work.

2. A Comparative Analysis of Functional Correctness

The relationship between programs and the mathematical functions they compute has long been of interest to computer scientists [McCarthy 63, Strachey 66]. More recently, [Mills 72, 75] 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 75, Misra 78] and now appears as a viable alternative to the inductive assertion verification method due to [Floyd 67, Hoare 69].

In this chapter, a tutorial view of the functional correctness theory is presented which is based on a set of structured programming control structures. An implication of this verification theory 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 light of the functional and inductive assertion theories.

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 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 2.1 - 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
$[P]=\{(<a,b>,<0,b*(a!)>) \mid a>=0\} \cup \{(<a,b>,<a,b>) \mid a<0\}$.
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 and 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].

2.1. The Functional Correctness Technique

We will make use of the following 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. If g is a function or binary relation, g**-1 represents the binary relation which corresponds to the inverse of g. We will 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.

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. We first need the following definition.

Definition 2.1 - 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 2.1 - If the loop B*H is closed for the domain of a function f, then the loop computes f iff, for all $X \in D(f)$
(2.1) the loop terminates when executed in

the initial state X,

(2.2) $B(X) \rightarrow f(X) = f(H(X))$, and

(2.3) $\neg B(X) \rightarrow f(X) = X$.

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 $0 \leq i < n$, we have $B(X[i]) \& X[i+1] = H(X[i])$ and also $\neg B(X[n])$. Condition (2.2) shows $f(X[0]) = f(X[1]) = \ldots = f(X[n])$. 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 $f(X) \neq f(H(X))$. From the closure requirement, $H(X) \in D(f)$ and the loop produces $f(H(X))$ when given the input $H(X)$. But this implies the loop can distinguish between the cases where $H(X)$ is an input and the case where $H(X)$ 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 $f(X) \neq X$. Thus the loop must not compute f.

An important aspect of Theorem 2.1 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 2.2 - Using the loop P and function f of Example 2.1, we shall show P computes f. D(f) is the set of all states satisfying $a \geq 0$. Since a is prevented from turning negative by the loop predicate, the loop is closed for D(f) and Theorem 2.1 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 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 in this chapter is based on prime program decomposition [Linger, Mills & Witt 79]. 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 2.4.

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 the domain of its intended function. These intended functions must either be supplied with the program or some heuristic (see Chapters 4-5) 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 functional correctness technique will be described by a set of function derivation rules and a set of function verification rules. These rules are given in Figure 2.1.

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 2.3 - 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.

```
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 2.1.
```

Figure 2.1 Derive and Verify Rules

Example 2.4 - 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 := v1+v3,v2) #v2,v3#

is equivalent to

(v1 > 0 -> v1,v2,v3 := ?,v1+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 <2,4,2> as well as <-2,4,2>. 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 function. 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 2.4, 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 (i.e. constant functions) may have an empty domain set.

Example 2.5 - 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, outer WHILE loop and inner WHILE loop respectively. The notation SUM(a,b,c,d) used in these functions stands for the summation from a=b to c of the expression d. We use the notation Fn-m as the derived function for lines n through m of the program.

Step 1) - Using derive rules D1 and D2 we get

F5-6 = j,p := j+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

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

and

j=n -> <p*a**(n-j),n> = <p,j>

which are true.

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

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

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

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

tions are (where n>=0)
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>=0 -> s := s+SUM(i,a,m,i**n))#s# o f1-1
      = (n>=0 -> s := s+SUM(i,a,m,i**n))#s# o a,s := 1,0
      = (n>=0 -> 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 72, Mills 75]. This verification method is compared and contrasted with the inductive assertion technique in [Basili & Noonan 80]. The 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.

The essential idea behind Theorem 2.1 can be traced to [McCarthy 62, McCarthy 63], in which a technique for proving two functions equivalent, "recursion induction," is described. Theorem 2.1 can be viewed as a specific application of this technique. In [Manna & Pnueli 70, Manna 71] and more recently [Morris & Wegbreit 77], loop verification rules similar to that stated in Theorem 2.1 are suggested. In [Basu & Misra 75], the authors prove a result which corresponds to Theorem 2.1 for the case where the loop contains local variables.

The closure requirement of Theorem 2.1 has received considerable attention. Several classes of loops which can be proved without the strict closure restriction are discussed in [Basu & Misra 76, Misra 78, Misra 79, Basu 80]. In [Wegbreit 77], however, the author describes a class of programs for which the problem of "generalizing" a loop specification in order to satisfy the closure requirement is NP-complete.

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

An important implication of Theorem 2.1 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 a loop which is closed for the domain of a function it computes.

A loop assertion for the loop B*H is a boolean-valued expression which 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

will denote by X), as well as 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 ∈ D

(2.4) A(X0,X0)

(2.5) A(X0,X) & B(X) -> A(X0,H(X)) & (H(X) ∈ D).

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 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 conditions which are valid when (and if) the execution of the loop terminates. Specifically, any assertion which can be inferred from

(2.6) A(X0,X) & ~B(X)

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. 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 (2.6). 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

(2.7) X0 ∈ D(f) & A(X0,X) & ~B(X) -> X=f(X0).

A loop invariant A(X0,X) over some set containing D(f) for which condition (2.7) holds is an f-adequate loop invariant.

Example 2.6 - Let P denote the program

```
while a ∉ {0,1} do
   if a > 0 then
      a := a - 2
   else a := a + 2 fi
od.
```

Consider the following predicates

A1(<a0>,<a>) <-> TRUE
A2(<a0>,<a>) <-> ABS(a) <= ABS(a0)
A3(<a0>,<a>) <-> ODD(a) = ODD(a0)
A4(<a0>,<a>) <-> ODD(a) = ODD(a0) & ABS(a) <= ABS(a0)
A5(<a0>,<a>) <-> ODD(a) = ODD(a0) OR (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)>) | a is an integer}, and consider A3. Since a ∈ {0,1} implies a=ODD(a), we can infer a=ODD(a0) from A3(<a0>,<a>) & a ∈ {0,1}. Thus A3 is an f-adequate loop assertion. Similarly, A4 and A5 are f-adequate loop assertions, but neither A1 nor A2 is restrictive enough to be f-adequate. Predicates A3 and A4 are loop invariants over D; however, since A5 fails (2.5), it is not a loop invariant (a=3,a0=2 is a counter example).

Theorem 2.2 - 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 if A(X0,X) is any f-adequate loop invariant over D(f), A(X0,X) -> f(X)=f(X0) for all X,X0 ∈ D(f).

Proof - First we show that f(X)=f(X0) is a loop invariant over D(f). Condition (2.4) is f(X0)=f(X0). From Theorem 2.1, for all X ∈ D(f),

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

Thus for all X,X0 ∈ D(f),

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

Adding the closure condition B(X) -> H(X) ∈ D(f) yields condition (2.5). Thus f(X)=f(X0) is a loop invariant over D(f). Again from Theorem 2.1, for all X ∈ D(f),

$$\tilde{}B(X) \rightarrow f(X)=X.$$

Thus for all X0 ∈ D(f),

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

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 ∈ D(f), there exists some sequence Z[1],Z[2], ... ,Z[n] (possibly with n=1) where Z[1]=Z, Z[n]=f(Z), ~B(Z[n]), and with B(Z[i]) & Z[i+1] = H(Z[i]) for all i satisfying 1 <= i < n. By condition (2.5) we have A(Z0,Z[1]), A(Z0,Z[2]), ... ,A(Z0,Z[N]); thus A(Z0,f(Z)) and ~B(f(Z)). Since Z0 ∈ D(f) and A(X0,X) is f-adequate,

$$A(Z0,f(Z)) \;\&\; \tilde{}B(f(Z)) \rightarrow f(Z0)=f(Z)$$

from condition (2.7). Thus for all Z0,Z ∈ D(f),

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

Example 2.6 (continued) - In this example, A3 is of the form f(X)=f(X0). A3 is clearly weaker than the other f-adequate loop invariant A4. It is worth noting that A3 is not weaker than A5, but A5 is not a loop invariant, and A3 is not weaker than A2, but A2 is not f-adequate. This situation is illustrated in Figure 2.2. The sets labeled S1-S5 are the sets of ordered pairs (<a0>,<a>) satisfying A1-A5, respectively, i.e.

$$SI=\{(<a0>,<a>) \mid AI(<a0>,<a>)\}$$

for I=1,2,3,4 and 5. The diagram is partitioned in half with a ∉ {0,1} on the left and a ∈ {0,1} on the right. Note that S4 (or the set corresponding to any f-adequate loop invariant for that matter) is a subset of S3. Furthermore, the set corresponding to each f-adequate loop assertion is identical where a ∈ {0,1}.

This region of the diagram is precisely the set f.

Consider the problem of using Hoare's iteration axiom

(2.8) I & B {X:=H(X)} I -> I {B*H} I & ~B

to prove 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 ∈ 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 I over exactly D(f), Theorem 2.2 tells us that f(X)=f(X0) is the

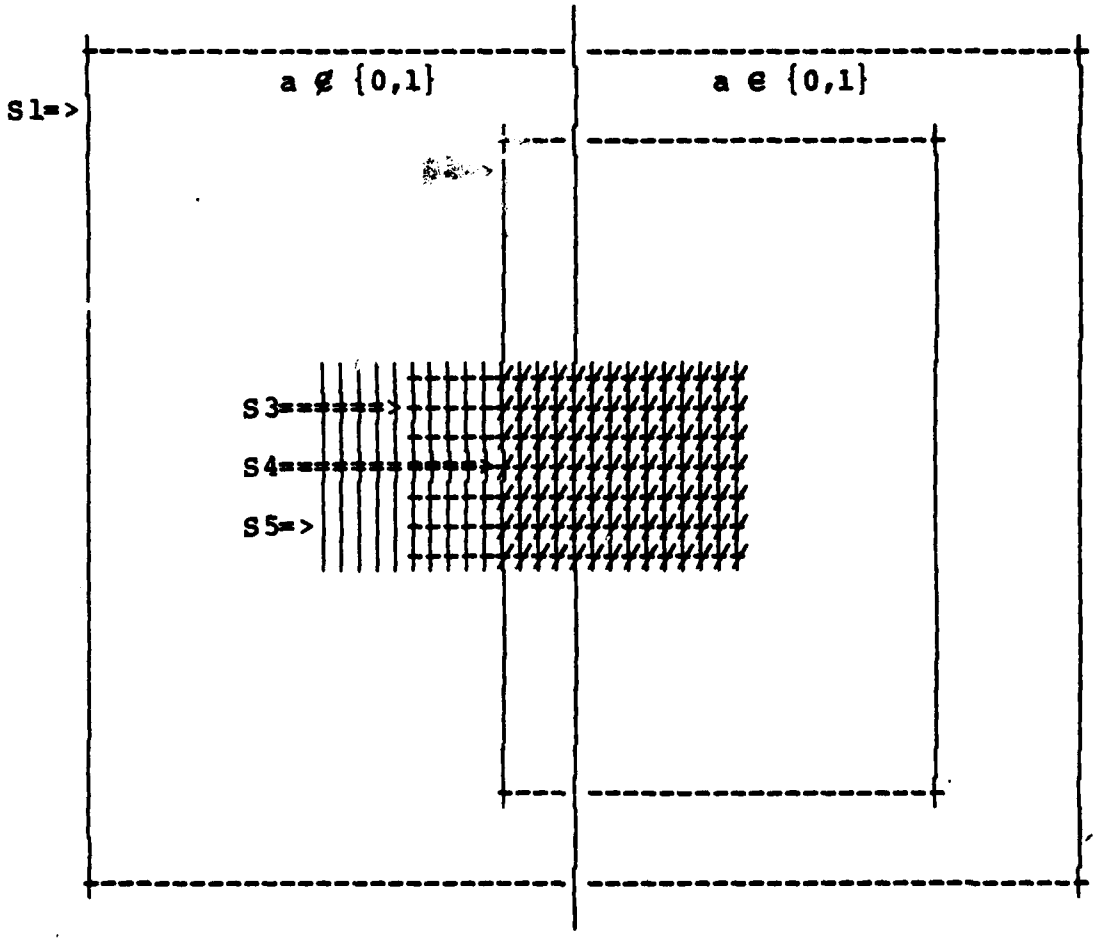


Figure 2.2 The Sets S1-S5

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 antecedent to (2.8) is true), because it says less (is less restrictive, is satisfied by more data states, etc.) than other loop invariants. Along these lines, one might conclude that if a loop is closed for the domain of a function f, Theorem 2.2 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*H compute f and let A(X0,X) be an f-adequate loop invariant. It is clear that in this case

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

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 cause B to evaluate to TRUE. 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. A4 in Example 2.6). However, Theorem 2.2 gives us a technique for constructing the only f-adequate invariant over D(f) that will be valid for <u>any</u> 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 of f. (S is a level set of f iff there exists a Y such that $S=\{X \mid f(X)=Y\}$). 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*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 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 2.7</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 2.2, we know that

A(<a0,b0,c0>,<a,b,c>) <-> <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. A(<4,10,7>,<4,10,7>), A(<4,10,7>,<3,10,17>), ... , A(<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 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<=a<=a0 is an internal invariant, but A(<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 can be proven 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[1]. However, if we consider nondeterministic loops and weaken the definition of an internal invariant to one where A(X0,X) implies 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 corresponds to a "blind search" implementation of the function.

In [Basu & Misra 75], the authors emphasize the difference between loop invariants and loop assertions. The fact that f(X)

---

[1] Outline of proof: Let f(Y)≠Y, B*H compute f and suppose 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.

= f(X0) is an f-adequate loop invariant appears in [Basu & Misra 75, Linger, Mills & Witt 79]. The independence of this loop invariant from the characteristics of the loop body is discussed in [Basu & Misra 75].

2.3. Comparing the Hoare and Mills Loop Proof Rules

An alternative to using Theorem 2.1 in showing a loop computes a function is to apply Hoare's inductive assertion verification technique. That is, one could verify P {B*H} Q where

P <-> X=X0 & X ∈ D(f), and
Q <-> X=f(X0)

by demonstrating the following for some predicate I:

A1: P -> I
A2: B & I {X:=H(X)} I
A3: ~B & I -> 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. Recalling from Theorem 2.1, if B*H is closed for D(f), the functional verification rules are:

F1: X ∈ D(f) -> B*H terminates for the input X
F2: X ∈ D(f) & B(X) -> f(X) = f(H(X))
F3: X ∈ D(f) & ~B(X) -> 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 2.3 - 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 <-> 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 <-> f(X)=f(X0) in the inductive assertion rules, then

A1 <-> TRUE,
A2 <-> F2 provided X ∈ D(f) & B(X) -> X ∈ D(H),
A3 <-> F3,
A4 <-> F1.

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

X=X0 & X ∈ D(f) -> f(X)=f(X0)

which is clearly true for any f. Combining with our first result yields A1 & A4 <-> 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': X,X0 ∈ D(f) & ~B(X) & f(X)=f(X0) -> X=f(X0).

Note that A3' -> F3 by considering the case where X=X0.

Furthermore, F3 -> A3' by considering the case where X0 ∈ D(f) and f(X)=f(X0). Now we have A3 <-> A3' <-> F3 and adding this to our result above we get A1 & A3 & A4 <-> F1 & F3. We next prove A2 & A4 <-> F1 & F2. This combined with the above equivalence yields the desired result A1 & A2 & A3 & A4 <-> F1 & F2 & F3. Note that if there exists an X ∈ D(f) such that B(X) 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 <-> F2 & F1. We now consider the other case where for all X ∈ D(f), B(X) -> X ∈ D(H). In this situation we will show A2 <-> F2; combining with A4 <-> F1 yields A2 & A4 <-> F2 & F1. 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 ∈ D(f) such that B(X) by hypothesis, this may be transformed using Hoare's axiom of assignment to the implication

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

As before, we can show A2'->F2 by considering the case where X=X0, and F2->A2' by considering the case where X0 ∈ 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 2.3 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," testing that the desired result can be implied from the predicate f(X)=f(X0) and the negation of the predicate B.

2.4. Subgoal Induction and Functional Correctness

Subgoal induction is a verification technique due to [Morris & Wegbreit 77]. It is based largely on work appearing in [Manna & Pnueli 70, Manna 71]. In this section we compare subgoal induction to the functional correctness approach described above.

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 the same problem) are sufficient and necessary conditions for correctness. Results in [Misra 77] 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 more precisely compare the two techniques, we consider the flow chart program in Figure 2.3 adapted from [Morris & Wegbreit 77]. In the figure, P1,P2,P3 and P4 are points of control in the flow chart, 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 flow chart program amounts to a WHILE loop surrounded by pre and post processing. Our goal is to prove the program computes a function f. Morris & 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, inductively shows it holds for all P2 to P4 execution paths, and then uses v to show 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: $X \in D(v) \;\&\; \neg B(X) \rightarrow v(X) = Q(X)$

S2: $X \in D(v) \;\&\; B(X) \rightarrow v(H(X)) = v(X)$

S3: $X \in D(f) \rightarrow f(X) = v(K(X))$.

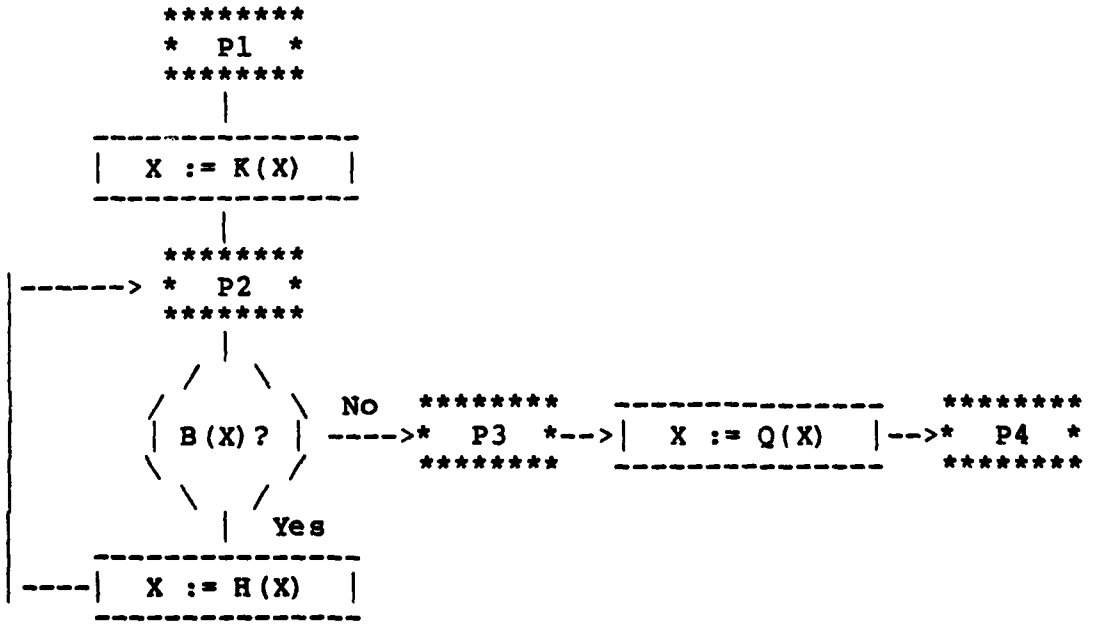


Figure 2.3 Flow Chart Program

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

The functional verification theory presented here is similar with the exception that the function Q is not included in the induction path. We select some function g and show it holds for all P2 to P3 execution paths (i.e. we show the WHILE loop computes g) and then use g to show f is computed by all P1 to P4 execution paths. Once g has been selected, the verification conditions are

F1: $X \in D(g)$ & $\neg B(X) \rightarrow g(X)=X$
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 the program computes f. The function Q 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

$$v = Q \circ g.$$

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

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

F1′: $X \in D(g)$ & $\neg B(X) \rightarrow g(X)=X$
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 nature of the functional correctness algorithm described in Section 2.1. 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

2.1.

Example 2.8 - We wish to show the program

```
while x ∉ {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={(<x>,<ODD(x)>)}. The subgoal induction verification conditions are

x ∈ {0,1,2,3} -> Q(x) = ODD(x), and
x ∉ {0,1,2,3} -> ODD(H(x)) = ODD(x), where

Q(x) = if x > 1 then x-2 else x, and
H(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

g = {(<x0>,<x>) | x ∈ {0,1,2,3} & x mod 4 = x0 mod 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 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 ∈ {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, provided the loop is closed for D(f), an inductive assertion 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.

2.5. 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 will again consider the program in Figure 2.3 with the understanding that Q is a null operation (identity function). We want to prove that the program computes a function f, i.e. that f holds for all P1 to P3 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) [Linger, Mills & Witt 79]. 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. rather than treating the initialization and the loop individually). In order for the program to compute f, it must be that K(X)=K(Y) -> f(X)=f(Y). Consequently, the relation represented by f o (K**-1) is a function and is a candidate for the intermediate hypothesis g. Indeed, the initialized loop program is correct with respect to f iff g = f o (K**-1) is a function and the WHILE loop (by itself) is correct wrt g. Unfortunately, the domain of this function is the image of D(f) through 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 g can be thought of as one of generalizing f o (K**-1).

Example 2.9 - We want to show the program

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

computes

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

As before, SUM(k,1,n,a[k]) is a notation for a[1]+a[2]+ ... +a[n]. If K represents the function performed by the initialization, f o (K**-1) is

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

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

g = s:=s+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 2.2 indicates that the invariant A(X0,X) <-> g(X)=g(X0) over D(g) can be used to show the loop computes g; absorbing the initialization X:=K(X) into the invariant gives the result that the limited invariant A(X0,X) <-> g(X)=g(K(X0)) can be used to prove the initialized loop program computes g o 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? We motivate the result as follows: we could obtain an equivalent program by modifying the initialization to (nondeterministically) map X0 to X if A(X0,X) is true. 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 that was used to show that the loop, assuming correctness, must compute f o (K**-1), the loop must also compute f o (A(X0,X)**-1). That is, if A(X0,X) holds for some X0 ∈ 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 one problem implies a solution to the other problem.

Example 2.9 (continued) - An inductive assertion proof of our program might use the limited invariant s=SUM(k,1,i,a[k]) & 0<=i<=n. Note that this invariant implies the invariant g(K(X0))=g(X) discussed above (where g and K are as defined previously). Using the technique outlined above, we may derive from this invariant the loop function

g' = (s=SUM(k,1,i,a[k]), 0<=i<=n -> s:=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 2.2 states that such an invariant must be g'-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(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.

The translation between loop invariants and intermediate hypotheses in a subgoal induction proof is discussed in [Morris & Wegbreit 77, King 80]. In Chapter 3, we propose a new verification strategy for initialized loop programs which is based on the above mentioned notion of a nondeterministic loop initialization.

2.6. Discussion

Our purpose in this chapter has been to explain the functional verification technique in light of other program correctness theories. The functional technique is based on Theorem 2.1 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 2.2, 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 2.3 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 based on prime program decomposition. In the following chapter, the idea of a proof transformation is introduced and is proposed as a decomposition strategy which overcomes this deficiency in prime program decomposition.

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 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.

In Chapter 4, we deal with the functional correctness requirement that each program loop be documented with an appropriate intended function. In order to alleviate this problem, a heuristic technique is proposed which can be used as an aid in ascertaining undocumented intended functions for program loops.

We explained in Section 2.5 that it is possible to obtain a loop function for a loop preceded by initialization based on the initialization and the program specification. This function, however, usually has a restricted domain and must then be generalized to meet the closure requirement of Theorem 2.1. In Chapter 5, we discuss a class of loops for which these generalizations may be obtained in a systematic manner.

3. A New Verification Strategy For Iterative Programs

The difficulties associated with the general verification of computer programs are well known. Chief among these difficulties is the problem of creating a suitable inductive hypothesis for programs which use iteration or recursion. A large number of guidelines or heuristics for the construction of these hypotheses have appeared in the literature. Particularly promising are results along the lines of Theorem 2.1 [Mills 75, Basu & Misra 75, Wegbreit 77, Morris & Wegbreit 77] which show that for a particular class of program/specification pairs, an appropriate inductive hypothesis may be obtained directly from the specification. The direct consequence of these results is that for a verification problem in this class, the program can be proven/disproven correct with respect to its specification (assuming termination) by testing several verification conditions based on the specification and characteristics of the program components. Unfortunately, this class is rather restricted and, as a result, many verification problems which arise naturally in practice are not covered by these results.

In this chapter, a verification strategy is described which is based on the idea of applying a correctness/incorrectness preserving transformation to the program under consideration. The motivation behind the transformation is to produce a verification problem such that, in a manner similar to that in the above mentioned work, an inductive hypothesis can be directly obtained. Thus we are proposing replacing the problem of synthesizing an inductive hypothesis with one of discovering an adequate correctness/incorrectness preserving transformation. In a number of examples we have studied, the latter problem appears to be more tractable.

In the remainder of this section we discuss a general verification problem which occurs often in practice and then suggest the idea of a transformation as a means to solve the problem. A number of heuristics for discovering an appropriate transformation are given in Section 3.1 and are illustrated with examples. In Section 3.2, the proposed solution is compared and contrasted with the inductive assertion and subgoal induction approaches to the verification problem. Finally, the application of the technique to more complex program forms is discussed in Section 3.3.

In our analysis, we will consider a program P of the form

```
X := K(X);
while B(X) do
    X := H(X)
od;
X := Q(X)
```

where X is the program data state, K, H and Q represent data-state-to-data-state functions and B is a predicate on the data

state. For the present we assume each of the functions and the predicate is explicitly known; this requirement regarding the function Q is relaxed in Section 3.3. We assume that the program specification is formulated as a function mapping the input data state to the portion of the output data state which corresponds to the program variables whose final values are of interest. We will use a function e which extracts from any data state the data state portion corresponding to these variables. Thus for this verification problem, P is correct with respect to its specification function f if and only if for any X in D(f), [P](X) is defined and f(X)=e([P](X)).

Let TAIL be the portion of P which follows the initialization X := K(X), i.e. TAIL is the WHILE loop followed by the assignment X := Q(X). We begin with the following:

Observation 1 - P is correct wrt f iff TAIL is correct wrt the input/output relation

$$g = \{(X,Y) \mid \nexists\, X0 \in D(f) \vdash (K(X0)=X \;\&\; f(X0)=Y)\}.$$

That is, P is correct wrt f iff on data states X which are output by K on input X0 ∈ D(f), TAIL produces f(X0). For the moment, we make the following two assumptions:

FUNCTION: The relation g is a function, i.e.
K(X0)=K(X1) -> f(X0)=f(X1)
for all X0, X1 ∈ D(f), and

CLOSURE: The WHILE loop is closed (see Section 2.1) for the domain of g, i.e.
K(X0)=X, B(X) -> ∄ X1 ∈ D(f) ⊢ K(X1)=H(X)
for all X0 ∈ D(f)[2].

Given these two assumptions, it is easy to state the necessary and sufficient verification conditions for the correctness of TAIL (assuming termination) wrt g. These are (adapted from [Mills 75, Basu & Misra 75, Morris & Wegbreit 77])

X ∈ D(g), B(X) -> g(X)=g(H(X))
X ∈ D(g), ~B(X) -> g(X)=e(Q(X))

and are called the iteration and boundary conditions, respectively, in [Misra 78]. They are equivalent to

X0 ∈ D(f), K(X0)=X, B(X) -> g(X)=g(H(X))
X0 ∈ D(f), K(X0)=X, ~B(X) -> g(X)=e(Q(X))

i.e.

X0 ∈ D(f), B(K(X0)) -> g(K(X0))=g(H(K(X0)))
X0 ∈ D(f), ~B(K(X0)) -> g(K(X0))=e(Q(K(X0)))

i.e.

---

[2] Throughout this dissertation, we frequently use the notation

P1, P2, ..., PN -> B

for the slightly longer

P1 & P2 & ... & PN -> B.

X0,X1 ∈ D(f), B(K(X0)), K(X1)=H(K(X0)) -> g(K(X0))=g(K(X1))
X0 ∈ D(f) ,~B(K(X0)) -> g(K(X0))=e(Q(K(X0)))

i.e.

ITERATION:
X0,X1 ∈ D(f), B(K(X0)), K(X1)=H(K(X0)) -> f(X0)=f(X1)

BOUNDARY:
X0 ∈ D(f) ,~B(K(X0)) -> f(X0)=e(Q(K(X0))).

We conclude as follows:

Observation 2 - Suppose FUNCTION and CLOSURE hold. P is correct wrt f (assuming termination) iff ITERATION and BOUNDARY hold.

Example 3.1 - Consider the following program

```
{a>=0}
a := a/2;
while a ∉ {0,1} do
   a := a - 2
   od;
a := (if a=0 then 1 else 0)
{a=EVEN(a0/2)}.
```

In this program, the data state consists of the single integer variable a and will be represented by <a>. The function EVEN(a) appearing in the program postcondition returns 1 if its argument is even and 0 otherwise. The specification function f for this program is

f(<a0>)=a <-> a0>=0 & a=EVEN(a0/2)

which implies

<a0> ∈ D(f) <-> a0>=0.

This program relates to the general program form above as follows

K(<a0>)=<a> <-> a=a0/2
B(<a>) <-> a ∉ {0,1}
H(<a0>)=<a> <-> a=a0-2
Q(<a0>)=<a> <-> (a0=0 & a=1) OR (a0≠0 & a=0)
e(<a>)=a.

Note that assumptions FUNCTION and CLOSURE hold. If the program terminates for all inputs in D(f), it is correct wrt its specification iff conditions ITERATION and BOUNDARY hold. These can be written

a0>=0, a1>=0, a0/2 ∉ {0,1}, a1/2=a0/2 - 2 -> EVEN(a0/2)=EVEN(a1/2)

and

a0>=0, a0/2 ∈ {0,1} -> EVEN(a0/2)=(if a0/2=0 then 1 else 0)

respectively.

At this point we stop to consider the assumptions we have made. Suppose assumption FUNCTION is false, but CLOSURE holds. Thus we have X0, X1 ∈ D(f) satisfying

K(X0)=K(X1) & f(X0)≠f(X1).

Now if B(K(X0)), then K(X2)=H(K(X0)) for some X2 ∈ D(f) by CLOSURE. If ITERATION was valid, we would have f(X0)=f(X2) as well

as f(X1)=f(X2). Since this contradicts our hypothesis, ITERATION does not hold. On the other hand, suppose ~B(K(X0)). If BOUNDARY held, we would have f(X0)=e(Q(K(X0))) as well as f(X1)=e(Q(K(X1))). The contradiction leads us to conclude that if FUNCTION is false, but CLOSURE holds, (at least) one of ITERATION and BOUNDARY is false. Since FUNCTION being false implies P is not correct wrt f (i.e. [P](X0)=[P](X1), but f(X0)≠f(X1)), this leads us to the following:

Observation 3 - Suppose CLOSURE holds. P is correct wrt f (assuming termination) iff ITERATION and BOUNDARY hold.

Unfortunately, it is much more difficult to deal with the reliance on assumption CLOSURE. We can, however, gain insight into the problem by studying CLOSURE and its relation to condition ITERATION. Imagine P executing on some input in D(f) and consider the sequence of intermediate data states on which the predicate B is evaluated. CLOSURE requires that each of these intermediate states be "reachable" through the function K for some input element of D(f). It is this reachability that enables condition ITERATION to test whether this sequence of states stays on the right "track," or more specifically, that the inverse images of these states through K remain in the same level set of f.

Most often in practice, however, the purpose of the initialization K is to "constrain" the data state, thereby providing some specific "starting point" for the loop execution. In this case, the intermediate data states will not be reachable through K and, as a result, condition ITERATION may well hold for the simple reason that it is vacuously true, i.e. the term K(X1)=H(K(X0)) will be false for all X1. Rather than abandoning this approach to the verification of P, however, the solution suggested here is to "correct" the problem and proceed.

Consider substituting for K a suitable replacement initialization K′. By "suitable" here, we mean that the output of K′ must be sufficiently unconstrained so that CLOSURE holds for K′, and furthermore, that the substitution preserves the termination and correctness/incorrectness properties of the program P. The following definition formalizes this idea.

Definition 3.1 - Let P′ be the program P with the initialization K replaced by K′. K′ is a reduction hypothesis iff CLOSURE holds for K′ and each of

TERMINATES: For inputs in D(f), P terminates -> P′ terminates, and

PRESERVES: P is correct wrt f iff P′ is correct wrt f

is satisfied.

The significance of the definition is that once a reduction hypothesis K′ has been located, we can prove/disprove the correctness of the original program by proving its termination

and verifying ITERATION and BOUNDARY with K′ substituted for K.

The proposed solution of finding a reduction hypothesis is analogous to finding an adequate loop invariant for an inductive assertion proof [Hoare 69], or finding an appropriate loop function for a functional [Mills 75] or subgoal induction [Morris & Wegbreit 77] proof. We justify proposing an alternative to these standard techniques for the following reasons:

a) a reduction hypothesis has a unique intuition behind it; in a number of cases, this leads rather naturally to a solution,
b) the class of reduction hypotheses bears an interesting relationship to the class of adequate loop invariants for P,
c) unlike the technique of inductive assertions, it is possible to disprove an incorrect program without considering the program beyond the loop, i.e. by disproving ITERATION,
d) in the case there is no initialization (i.e. K is the identity function) and the loop is closed for D(f) (or, more generally, any time CLOSURE holds for K), a very efficient proof results since K itself is a reduction hypothesis, and
e) the reduction hypothesis solution provides continuity between the cases where CLOSURE does and does not hold; an understanding of "why" CLOSURE does not hold, for example, can provide insight into how to create K′ from K.

3.1. Constructing a Reduction Hypothesis

In this section we will consider several heuristics for creating a reduction hypothesis for P and will illustrate their use on example programs. We begin with some preliminary remarks.

In the discussion in the preceding section, we assumed that the program pieces corresponding to K, H and Q were deterministic, that is, we assumed their semantics (i.e. their input/output behavior) could be represented by data-state-to-data-state functions. The above results, however, extend in the natural way to the case where subprograms in P are nondeterministic provided one switches to a slightly more awkward relational notation for representing these subprograms. In particular, we will be interested in the case where the initialization is nondeterministic since it turns out that this kind of initialization is often a reasonable choice for a reduction hypothesis. If this new initialization is represented by a data-state-to-data-state relation K′, ITERATION and BOUNDARY translate to

ITERATION′:
X0,X1 ∈ D(f), K′(X0,X), B(X), K′(X1,H(X)) -> f(X0)=f(X1)
BOUNDARY′:
X0 ∈ D(f), K′(X0,X), ~B(X) -> f(X0)=e(Q(X)),

where K′(X0,X) is a notation we will use which stands for (X0,X)

$\in K'$.

Now suppose this relation K' satisfies CLOSURE, i.e.

$$K'(X0,X) \;\&\; B(X) \rightarrow \exists\, X1 \in D(f) \vdash K'(X1,H(X))$$

for all $X0 \in D(f)$. The program P' derived from P by replacing the initialization K with K', i.e. by replacing

```
X := K(X)
```

with

```
X := "any Y satisfying K'(X,Y)",
```

is correct wrt f (assuming termination) iff ITERATION' and BOUNDARY' hold. If K' has been chosen so as to be a reduction hypothesis, the original P is correct wrt f iff P terminates for inputs in D(f) and ITERATION' and BOUNDARY' hold.

As an aid in expressing nondeterministic program segments, we will use a notation defined in [Dijkstra 76]. Specifically, an execution of the program

```
if B1(X) -> P1
 | B2(X) -> P2
   .
   .
   .
 | Bn(X) -> Pn
fi
```

calls for the execution of any single program Pi provided guard Bi(X) holds.

We now consider two opposing alternatives to constructing K'. Each is based on a different philosophy for insuring that PRESERVES is satisfied. The first is a program-oriented approach and is characterized by selecting K' so that the programs P and P' are equivalent, i.e. so that P and P' exhibit identical input/output behavior. Such a K' trivially guarantees PRESERVES is satisfied. The other approach is a specification-oriented approach and is characterized by selecting K' as a superset of K in such a way that executions of P' which use the extended aspect of the initialization are guaranteed to be correct, i.e. execute in accordance with the program specification. Such a K' satisfies PRESERVES since the correctness of P' implies the correctness of P (since K' is a superset of K) and the correctness of P implies the correctness of P' (the additional execution paths in P' are known to be correct). Thus each approach chooses a different technique aimed at meeting PRESERVES. If, in addition, CLOSURE holds for this new initialization and TERMINATES is satisfied, then K' is a reduction hypothesis.

We will begin by considering several program-oriented heuristics and will make use of the following definition.

Definition 3.2 - Let G be a data-state-to-data-state relation and let P' be the program P with initialization K replaced by G. G is an alternative initialization to K iff P and P' have identical input/output behavior for inputs in both D(f) and D(G).

Thus if G is an alternative initialization to K, then, for a restricted set of inputs, G can be used in place of K without affecting the externally observable behavior of P. In what follows, we will make use of the following properties which are derived trivially from this definition:

- K is an alternative initialization to itself,
- the union of any number of alternative initializations is an alternative initialization, and
- any subset of an alternative initialization is an alternative initialization.

Our interest in alternative initializations is due to the following theorem.

Theorem 3.1 - Any alternative initialization G whose domain includes D(f) and which satisfies CLOSURE is a reduction hypothesis.

Proof - Let G meet the conditions stated in the theorem and let P′ stand for the program P with initialization G substituted for K. From the definition of an alternative initialization, P and P′ have identical input/output behavior over D(f). Thus TERMINATES and PRESERVES are trivially satisfied and G is a reduction hypothesis.

The implication of Theorem 3.1 is that a reduction hypothesis may be created by discovering a suitable alternative initialization. This is the basis for all program-oriented approaches to synthesizing a reduction hypothesis. The following theorem suggests three techniques for constructing alternative initializations.

Theorem 3.2 (Program-Oriented Heuristics) - Let K1, K2 and K3 be any functions satisfying

LONGCUT: K1(X)=Y -> B(Y) & H(Y)=K(X)
SHORTCUT: K2(X)=Y -> B(K(X)) & Y=H(K(X))
NOCUT: K3(X)=Y -> B(K(X)) & B(Y) & H(Y)=H(K(X))

for all X ∈ D(f). Then each of K1, K2 and K3 is an alternative initialization.

Proof - We will make repeated use of the WHILE loop property

B(X) -> [TAIL](X)=[TAIL](H(X)).

Let X ∈ D(f) and X ∈ D(K1). Then

[TAIL](K1(X))=[TAIL](H(K1(X)))=[TAIL](K(X))=[P](X),

hence K1 is an alternative initialization. Let X ∈ D(f) and X ∈ D(K2). Then

[TAIL](K2(X))=[TAIL](H(K(X)))=[TAIL](K(X))=[P](X),

hence K2 is an alternative initialization. Finally, let X ∈ D(f) and X ∈ D(K3). Then

[TAIL](K3(X))=[TAIL](H(K3(X)))=[TAIL](H(K(X)))
=[TAIL](K(X))=[P](X),

hence K3 is also an alternative initialization.

The labels on the conditions in Theorem 3.2 are motivated by the effect replacing K with the alternative initializations has on the number of iterations of the WHILE loop. K1 causes the loop to execute 1 additional iteration, K2 saves the loop an iteration, and K3 has no effect on the number of iterations. The significance of the theorem is that any combination of K, K1, K2 and K3 whose domain includes D(f) and which satisfies CLOSURE is a reduction hypothesis.

We now define a specification-oriented heuristic for creating a reduction hypothesis.

Theorem 3.3 (Specification-Oriented Heuristic) - Let G be any alternative initialization to K whose domain includes D(f) and let K4 be any function satisfying

VERYSHORTCUT: K4(X)=Y -> ~B(Y) & e(Q(Y))=f(X).

Let K' be the union of G and K4 and suppose K' satisfies CLOSURE. Then K' is a reduction hypothesis.

Proof - Let K' be as stated in the theorem and let P' stand for the program P with initialization K' substituted for K. Let X ∈ D(f) and suppose P terminates for input X. If P' executes on X and the aspect of the initialization K' from G is used, P' must also terminate since G is an alternative initialization to K; if the aspect of the initialization K' from K4 is used, P' must terminate since the output of K4 causes B to evaluate to false. Hence in any case, P' terminates on the input X and TERMINATES holds. The remainder of the proof consists of showing that PRESERVES is satisfied. Suppose P is correct wrt f. Let X ∈ D(f). If P' executes on X and the aspect of the initialization K' from G is used, then P' is also correct on the input X since G is an alternative initialization to K; if the aspect of the initialization K' from K4 is used, then P' must produce

e([TAIL](K4(X)))=e(Q(K4(X)))=f(X),

hence P' is again correct on the input X. Thus in any case, the program P' is correct wrt f. To show the converse, suppose P' is correct wrt f. We must show that this implies P is correct wrt f. Let X ∈ D(f). Then X ∈ D(G). By the correctness of P' and the fact that K' includes G, the program P'' derived from P by replacing the initialization K with G produces f(X) for the input X. Since G is an alternative initialization to K, P and P'' exhibit identical input/output behavior for X, hence P produces f(X) for the input X and thus P is correct wrt f. This completes the proof of the theorem.

As in Theorem 3.2, the label on the condition in Theorem 3.3 is used to suggest the effect the initialization K4 has on the execution of the program. The output of a function satisfying VERYSHORTCUT causes the predicate B to evaluate to false, and consequently, the WHILE loop in P will execute zero times. We repeat that the functions K1, K2 and K3 represent program-oriented heuristics since they are designed specifically to preserve the effect of the program P (possibly over a restricted

domain). Function K4, on the other hand, represents a specification-oriented heuristic since its purpose is to insure program behavior which is in agreement with the program specification. Now that these heuristics have been defined, we will illustrate their use with a number of examples.

Example 3.2 - This example illustrates a circumstance in which assumption CLOSURE "almost" holds but "not quite." In this situation it may be clear how to "expand" the initialization in order to satisfy CLOSURE in such a way that preserves the effect of the program. The program

```
{a>=0}
a := a*2;
while a>0 do
   a := a - 1;
   b := b + 1
   od;
a := b
{a=a0*2 + b0}
```

does not satisfy CLOSURE since only even values of a are output from the initialization K and the WHILE loop is not closed for this set. This problem could be "fixed" by supplementing K with another initialization, selected nondeterministically, which produced odd values of a. The combined effect would be an initialization which was capable of producing all (nonegative) values of a and would consequently satisfy CLOSURE. Based on this reasoning, we could supplement K with the initialization

a := a*2 + 1,

but an inspection of the loop-body text indicates that we must compensate in this case by subtracting 1 from b if the effect of the program is to be preserved. This is actually an application of heuristic LONGCUT of Theorem 3.2; specifically, the initialization

K1(<a0,b0>)=<a,b> <-> a=a0*2+1 & b=b0-1.

satisfies LONGCUT and is thus (by Theorem 3.2) an alternative initialization. In the above notation for nondeterminacy, the combined initialization may be written

```
if TRUE -> a := a*2
 | TRUE -> a := a*2 + 1; b := b-1
fi.
```

and the relation K' which represents this initialization is

K'(<a0,b0>,<a,b>) <-> ((a=a0*2 & b=b0) OR (a=a0*2+1 & b=b0-1)).

Since K' is the union of two alternative initializations (K and K1), K' itself is an alternative initialization, and thus, applying Theorem 3.1, it is a reduction hypothesis. Hence the program can be verified correct with respect to its specification (assuming termination) by showing that ITERATION' and BOUNDARY' hold. Condition ITERATION' has the following two aspects, which correspond to the cases where a is even and odd, respectively:

a0>=0, a1>=0, a=a0*2, b=b0, a>0, a1*2+1=a-1, b1-1=b+1
-> a0*2+b0=a1*2+b1

and

a0>=0, a1>=0, a=a0*2+1, b=b0-1, a>0, a1*2=a-1, b1=b+1
-> a0*2+b0=a1*2+b1.

These can be simplified (by eliminating a0, b0, a1 and b1) to

a>0 -> a+b=(a-2)+(b+2)

and

a>0 -> (a-1)+(b+1)=(a-1)+(b+1),

both of which hold. Condition BOUNDARY' is

a0>=0, a=a0*2, b=b0, a<=0 -> a0*2+b0=b

which simplifies to

a=0 -> a+b=b,

and thus also holds.

Example 3.3 - We noted above that the purpose of loop initialization is often to set the data state to some specific "starting point" for the execution of the loop. Once the loop begins execution, the data state leaves this starting point and takes on more general values. With this in mind, we note that the program P is equivalent to the program

```
X := H(K(X));
while B(X) do
   X := H(X)
   od;
X := Q(X)
```

on executions of P which require at least one loop iteration. Since the initialization in this new program includes an execution of the loop body, the hope is that the output of the new initialization will be general enough to satisfy CLOSURE. This observation is the motivation behind heuristic SHORTCUT in Theorem 3.2 and can be applied as follows: convert (if necessary) the verification problem to one over a domain where the loop will execute at least once and try the initialization H o K for the reduction hypothesis K'. As an illustration, consider the program

```
pa := 0;
while s ≠ NULL do
   pa := pa + head(s);
   s := tail(s)
   od;
if pa <= 0 then pa := 0 else pa := 1 fi
{pa=PAVG(s0)}
```

which determines whether the arithmetic average of the integers appearing in the sequence s is positive or negative. The function head(s) for a nonempty sequence s returns the lead element in s, tail(s) returns s with head(s) removed, and NULL denotes the sequence containing 0 elements. The function PAVG(s0) appearing in the postcondition has the value 1 if the average of the elements of s0 is positive and 0 otherwise. CLOSURE is not

satisfied for this program since pa takes on values other than 0 as the loop iterates. Suppose we can convince ourselves that the program executes correctly when the input s is the empty sequence and the loop is bypassed. The remainder of the proof then consists of verifying the program assuming the precondition {s≠NULL}. Accordingly, we will now use the program specification function

f(<pa0,s0>)=pa <-> s0≠NULL & pa=PAVG(s0).

The heuristic suggested above is to try the initialization

K2(<pa0,s0>,<pa,s>) <-> s0≠NULL & pa=head(s0) & s=tail(s0)

as a reduction hypothesis. This function satisfies SHORTCUT and is thus an alternative initialization. Its domain includes D(f) (i.e. {<pa,s> | s≠NULL}) and since it satisfies CLOSURE, it is a reduction hypothesis. To apply this result in a demonstration of the correctness of the program, we must show ITERATION′ and BOUNDARY′ hold (using K2 for K′). Condition ITERATION′ is

s0≠NULL, s1≠NULL, pa=head(s0), s=tail(s0), s≠NULL,
pa+head(s)=head(s1), tail(s)=tail(s1) -> PAVG(s0)=PAVG(s1)

which simplifies to

s≠NULL -> PAVG(<pa>||s)=PAVG(<pa+head(s)>||tail(s)).

where <x> is the sequence containing the single element x and || denotes concatenation of sequences. Condition BOUNDARY′ is

s0≠NULL, pa=head(s0), s=tail(s0), s=NULL
-> PAVG(s0)=POSITIVE(pa)

where POSITIVE(pa) is a function which has the value 1 if pa is positive and 0 otherwise. This condition simplifies to

s0=<pa> -> PAVG(s0)=POSITIVE(pa).

Example 3.4 - We now illustrate an application of the specification-oriented heuristic VERYSHORTCUT of Theorem 3.3. Since any output Y of a function K4 satisfying VERYSHORTCUT must satisfy ~B(Y), this approach is most appropriately used when CLOSURE holds for all loop iterations except the last, i.e. the only lack of values in the range of K is where B is FALSE. The idea is to add to the initialization the production of these values in such a way that assures the correct execution of the program. The following program serves to illustrate this kind of technique. It searches a linked list for an element containing a key field with the value 17:

```
found := FALSE;
while p ≠ NIL & ~found do
   if p^.key=17 then
      found := TRUE
   else p := p^.link fi
   od
{found=INCHAIN(p0,17)}
```

The program notations p^.key and p^.link are the key and link fields, respectively, of the node pointed to by p. A link field of NIL is used to mark the end of the list. The function INCHAIN(p0,17) appearing in the postcondition is a predicate which holds iff the chain pointed to by p0 contains a node with a

key field of 17. Note that since there is no program text following the loop, Q is the identity function. CLOSURE is not satisfied for this program since the variable found may take on the value TRUE on the last loop iteration. The initialization is extended by alternatively assigning the value TRUE to found, but only in the case where INCHAIN(p,17) holds. Specifically, we propose to supplement the given initialization with the function

K4(<found0,p0>)=<found,p> <-> INCHAIN(p0,17) & p=p0 & found

to yield the new initialization

```
if TRUE -> found := FALSE
[] INCHAIN(p,17) -> found := TRUE
fi.
```

Note that K4 satisfies VERYSHORTCUT, i.e. on executions of the program which follow the second path, the loop body is bypassed and the program necessarily behaves in accordance with its specification. The combined initialization is represented by the input/output relation

K'(<found0,p0>,<found,p>) <->
(~found OR found=INCHAIN(p0,17)) & p=p0.

Since arbitrary values of found and p may emerge from this initialization, CLOSURE is satisfied. By Theorem 3.3 (using the original initialization K for G), K' is a reduction hypothesis. We now use K' to state the necessary and sufficient conditions (assuming termination) for the correctness of the program. Corresponding to the two paths through the loop body of this program, there are two ITERATION' conditions

(~found OR found=INCHAIN(p0,17)), p=p0, p≠NIL, ~found,
p^.key=17, (~TRUE OR TRUE=INCHAIN(p1,17)), p=p1
-> INCHAIN(p0,17)=INCHAIN(p1,17)

and

(~found OR found=INCHAIN(p0,17)), p=p0, p≠NIL, ~found,
p^.key≠17, (~FALSE OR FALSE=INCHAIN(p1,17)), p^.link=p1
-> INCHAIN(p0,17)=INCHAIN(p1,17),

which simplify to

~found, p≠NIL, p^.key=17, INCHAIN(p,17)
-> INCHAIN(p,17)=INCHAIN(p,17)

and

~found, p≠NIL, p^.key≠17
-> INCHAIN(p,17)=INCHAIN(p^.link,17),

respectively. Condition BOUNDARY' is

(~found OR found=INCHAIN(p0,17)), p=p0, (p=NIL OR found)
-> found=INCHAIN(p0,17),

which simplifies to

(found & INCHAIN(p,17)) OR (~found & p=NIL)
-> found=INCHAIN(p,17).

Example 3.5 - This example illustrates a circumstance where several of the above heuristics are applied in order to create a reduction hypothesis. The following program sums the elements of a sequence:

```
{s≠NULL}
sum := 0;
while s≠NULL do
    sum := sum + head(s);
    s := tail(s)
    od;
{sum=SUM(s0)}.
```

The notation SUM(s0) appearing in the postcondition stands for the summation of the elements of s0. We begin our reasoning as follows. CLOSURE is not satisfied for this program because sum is constrained to the value 0 by an assignment statement. What would be the effect on the program of removing this assignment? After the first loop iteration, sum would have the value sum0+head(s0), rather than head(s0). We could compensate for this discrepancy by subtracting sum0 from head(s0) before the loop begins execution. Thus we choose to replace "sum:=0" with "head(s):=head(s)-sum." This is an application of heuristic NOCUT of Theorem 3.2. Indeed, the function corresponding to this new initialization,

K3(<sum0,s0>)=<sum,s>) <-> s0≠NULL & s≠NULL & sum=sum0 &
head(s)=head(s0)-sum0 & tail(s)=tail(s0),

satisfies NOCUT and is thus an alternative initialization. We note that CLOSURE is satisfied for this initialization on all but the final loop iteration (i.e. K3 cannot produce an output which satisfies s=NULL). As in Example 3.4, the solution under these circumstances is to supplement the initialization with a function satisfying VERYSHORTCUT; in this example such a function would be

K4(<sum0,s0>)=<sum,s>) <-> sum=SUM(s0) & s=NULL.

Note that the output of K4 causes B to evaluate to FALSE and forces the program to be correct wrt its specification. The complete initialization is then

```
if s≠NULL -> head(s) := head(s) - sum
[] TRUE  -> sum := SUM(s); s := NULL
fi.
```

Let K´ represent this initialization (i.e. let K´ be the union of K3 and K4). By Theorem 3.3 (using K3 for G), K´ is a reduction hypothesis. Corresponding to whether K3 or K4 is used in the term K´(X1,H(X)), condition ITERATION´ has the following two aspects:

```
s0≠NULL, sum=sum0, head(s)=head(s0)-sum0, tail(s)=tail(s0),
  s≠NULL, s1≠NULL, tail(s)≠NULL, sum+head(s)=sum1,
  head(tail(s))=head(s1)-sum1, tail(tail(s))=tail(s1)
            -> SUM(s0)=SUM(s1)
and
s0≠NULL, sum=sum0, head(s)=head(s0)-sum0, tail(s)=tail(s0),
  s≠NULL, sum+head(s)=SUM(s1), tail(s)=NULL
            -> SUM(s0)=SUM(s1)
which simplify to
s≠NULL, tail(s)≠NULL ->
            sum+head(s)=SUM(tail(s))=
            sum+head(s)+head(tail(s))+SUM(tail(tail(s)))
and
```

tail(s)=NULL -> sum+head(s)+SUM(tail(s))=sum+head(s).
Condition BOUNDARY' is
sum=SUM(s0), s=NULL -> SUM(s0)=sum.

3.2. Relation to Standard Correctness Techniques

In this section, we discuss the relationship between the proposed verification strategy and the subgoal induction [Morris & Wegbreit 77] (see also [Manna & Pnueli 70, Manna 71]), and inductive assertion [Hoare 69] correctness techniques. We will define these methods in the framework of the verification problem described above, i.e. in each case we wish to prove/disprove the program P correct wrt its specification function f.

All three techniques call for creating and verifying an hypothesis concerning some aspect of the behavior of P and then applying the hypothesis to prove/disprove the correctness of the program. In the proposed technique, this hypothesis is a reduction hypothesis; in subgoal induction, we will refer to the hypothesis as a tail function; in the inductive assertion technique, the hypothesis is an adequate inductive assertion.

A tail function g in a subgoal induction proof is a general description of the input/output behavior of program TAIL. Specifically, g has the same functionality as the specification f and must satisfy each of

SI1: X ∈ D(g), B(X) -> H(X) ∈ D(g)
SI2: X ∈ D(g), B(X) -> g(X)=g(H(X))
SI3: X ∈ D(g), ~B(X) -> g(X)=e(Q(X))
SI4: X ∈ D(g) -> TAIL terminates with input X
SI5: X ∈ D(f) -> K(X) ∈ D(g).

The first four of these conditions establish that TAIL is correct wrt g, SI5 assures that D(g) is sufficient for testing the correctness of the program P. When such a function g has been found, the program P is correct wrt f iff

SI6: X ∈ D(f) -> g(K(X))=f(X).

In the inductive assertion technique, an adequate inductive assertion is a sufficiently strong invariant relation between initial data states and data states occurring at the loop predicate B. For our purposes, an adequate inductive assertion for the program P is defined as a binary relation A over the data state of P which satisfies

IA1: X ∈ D(f) -> A(X,K(X))
IA2: X ∈ D(f), A(X,Y), B(Y) -> A(X,H(Y))
IA3: X ∈ D(f), A(X,Y) -> TAIL terminates with input Y
IA4: X ∈ D(f), A(X,Y), ~B(Y), A(X,Z), ~B(Z)
-> e(Q(Y))=e(Q(Z)).

If we view X and Y as representing the initial and current data states, IA1 and IA2 prove that A(X,Y) is a "loop invariant," IA3 is the necessary termination condition based on A, and IA4 tests whether A is sufficiently strong to verify the correctness of the program. When such a relation A has been found, the program P is

correct wrt f iff
IA5: X ∈ D(f), A(X,Y), ~B(Y) -> f(X)=e(Q(Y)).

The results of this section are contained in the following two theorems. They define a relationship between the three forms of program hypotheses and give a technique for transforming tail functions and adequate inductive assertions into reduction hypotheses.

Theorem 3.4 - Let g be a tail function for a subgoal induction proof of P, and let K' be a relation defined by
K'(X0,X) <-> g(K(X0))=g(X).
Then K' is a reduction hypothesis.

Proof - We must prove that K' satisfies CLOSURE and that TERMINATES and PRESERVES hold where P' is P with initialization K' substituted for K. To see that K' satisfies CLOSURE, let
X0 ∈ D(f) & K'(X0,X) & B(X)
i.e.
X0 ∈ D(f) & g(K(X0))=g(X) & B(X).
By SI1 and SI2, g(X)=g(H(X)); hence g(K(X0))=g(H(X)) and
X0 ∈ D(f) & K'(X0,H(X)),
thus CLOSURE holds for K'. TERMINATES must be satisfied since the range of K' contains only elements of D(g) and SI4 is satisfied. Finally, to show that PRESERVES holds, we will prove that P and P' compute the same function (in the variables of interest) over the domain D(f). Let X0 ∈ D(f). On input X0, P produces a result Y satisfying
K(X0)=X & Y=e([TAIL](X))
for some state X. On input X0, P' produces a result Y' satisfying
K'(X0,X') & Y'=e([TAIL](X'))
for some state X'. These may be rewritten
K(X0)=X & Y=g(X)
and
g(K(X0))=g(X') & Y'=g(X')
using the definition of K' and the fact that SI1-SI4 imply TAIL computes g. These imply
Y=g(K(X0))=Y',
hence P and P' compute the same function over D(f).

Thus a tail function g together with program initialization K can be used to construct a reduction hypothesis. The (perhaps) surprising result in the following theorem is that an adequate inductive assertion (by itself) is a reduction hypothesis, i.e. the class of reduction hypotheses for P contains the class of adequate inductive assertions for P.

Theorem 3.5 - If A is an adequate inductive assertion for P, then A is a reduction hypothesis for P.

Proof - We must show A satisfies CLOSURE and that TERMINATES and PRESERVES hold where P' is P with initialization A

substituted for K. To see that A satisfies CLOSURE, let

X0 ∈ D(f) & A(X0,X) & B(X).

By IA2, A(X0,H(X)), thus CLOSURE holds for A. TERMINATES follows directly from IA3. Finally, to show that PRESERVES holds, we will show that P and P′ compute the same function (in the variables of interest) over the domain D(f). Let X0 ∈ D(f). On input X0, P will produce Y satisfying

K(X0)=X & Y=e([TAIL](X))

for some state X. By IA1, A(X0,X). By IA3 the WHILE loop terminates on input X giving some result T where ~B(T). Repeated application of IA2 and the loop property

B(Z) -> [TAIL](Z)=[TAIL](H(Z))

gives A(X0,T) and [TAIL](X)=[TAIL](T)=Q(T). Thus on input X0, P produces Y satisfying

(3.1) A(X0,T) & ~B(T) & Y=e(Q(T))

for some T. P′ on the other hand, with input X0 will produces Y′ satisfying

A(X0,X′) & Y′=e([TAIL](X′))

for some state X′. Again by IA3, the WHILE loop terminates on input X′ giving some result T′ where ~B(T′). Repeated application of IA2 and the above loop property gives A(X0,T′) and [TAIL](X′)=[TAIL](T′)=Q(T′). Thus P′ produces Y′ satisfying

(3.2) A(X0,T′) & ~B(T′) & Y′=e(Q(T′))

for some T′. In light of IA4, (3.1) and (3.2) imply Y=Y′, thus P and P′ compute the same function over D(f).

Example 3.6 - To illustrate these ideas, we consider again the program discussed in Example 3.2:

```
{a>=0}
a := a*2;
while a>0 do
   a := a-1;
   b := b+1
   od;
a := b
{a=a0*2 + b0}.
```

The following are possible tail functions for a subgoal induction proof of the program

g1(<a0,b0>)=a <-> a0>=0 & a=a0+b0

g2(<a0,b0>)=a <-> a=MAX(a0,0)+b0.

Several adequate inductive assertions are

A1(<a0,b0>,<a,b>) <-> b=b0+2*a0-a & a>=0

A2(<a0,b0>,<a,b>) <-> b=b0+2*a0-a & a>=0 & a0>=0

A3(<a0,b0>,<a,b>) <-> b=b0+2*a0-a & a0>=a>=0

A4(<a0,b0>,<a,b>) <-> b=b0+2*MAX(a0,0)-MAX(a,0).

By Theorem 3.4,

K1(<a0,b0>,<a,b>) <-> g1(K(<a0,b0>))=g1(<a,b>)

and

K2(<a0,b0>,<a,b>) <-> g2(K(<a0,b0>))=g2(<a,b>),

i.e.

K1(<a0,b0>,<a,b>) <-> 2*a0>=0 & a>=0 & 2*a0+b0=a+b

and

K2(<a0,b0>,<a,b>) <-> MAX(2*a0,0)+b0=MAX(a,0)+b

are reduction hypotheses. Theorem 3.5 states that each of A1-A4 is also a reduction hypothesis. Thus any of these can be used in place of K' in the proof in Example 3.2. We remark that the relation K' used in that example is not an inductive assertion, hence the class of adequate inductive assertions is a proper subset of the class of reduction hypotheses.

3.3. Proof Transformations

Applying the verification technique proposed above based on ITERATION' and BOUNDARY' requires ascertaining the input/output behavior of the loop initialization, loop body and the program text following the loop. In many cases, this ascertaining process may be difficult (e.g. if these program segments contain additional WHILE loops). In view of this problem, we now consider the situation which occurs when the input/output behavior of the program text following the WHILE loop is not explicitly known. The verification problem under consideration, then, will be of the form

```
{X0 ∈ D(f) & X=X0}
X := K(X);
while B(X) do
   X := H(X)
   od;
T
{e(X)=f(X0)}
```

where K, B and H are as before and T represents some unspecified block of program text. Again our intention is to prove/disprove the program correct wrt its specification function f.

Suppose we have a reduction hypothesis K' for P and can show condition ITERATION' holds. What sense does BOUNDARY' make and how can we proceed? In this situation, condition BOUNDARY' corresponds to a new but simpler correctness problem. Specifically, we must prove

```
{X0 ∈ D(f) & K'(X0,X) & ~B(X)}
T
{e(X)=f(X0)}.
```

This problem is simpler than the original due to the fact that the loop has been eliminated from the program. Thus we have used a reduction hypothesis and condition ITERATION' to transform the correctness question for the original program P to a correctness question for a substantially simpler program. If the program T contains further looping structures, the process may be repeated.

Example 3.7 - The following program operates on sequences a, x, y and z of natural numbers. The function head(s) of a nonempty sequence s is the leftmost element of s, and tail(s) is s with head(s) removed. The infix operator || denotes

concatenation of sequences and NULL denotes the sequence with 0 elements. The predicate odd(a) is true iff a is odd.

```
while a ≠ NULL do
   if odd(head(a)) then y := y || <head(a)>
                   else z := z || <head(a)> fi;
   a := tail(a)
   od;
while y ≠ NULL do
   x := x || <head(y)>; y := tail(y)
   od;
while z ≠ NULL do
   x := x || <head(z)>; z := tail(z)
   od
{x=F(x0,y0,z0,a0)}
```

The function F appearing in the postcondition is defined as
F(x,y,z,a) = x || y || ODDS(a) || z || EVENS(a)
where ODDS(a) is the sequence containing the odd elements of a in the order they appear in a (EVENS(a) is similar). In this example, T corresponds to the last two WHILE loops and since there is no initialization, K is the identity function. Since CLOSURE holds for this function, K itself is a reduction hypothesis. Corresponding to the two paths through the first loop body, there are the two ITERATION' conditions, namely

a≠NULL, odd(head(a))
-> F(x,y,z,a)=F(x,y||<head(a)>,a,tail(a))

a≠NULL, ~odd(head(a))
-> F(x,y,z,a)=F(x,y,z||<head(a)>,tail(a)).

Once these have been proven, the verification problem then transforms to

```
{a=NULL}
while y ≠ NULL do
   x := x || <head(y)>; y := tail(y)
   od;
while z ≠ NULL do
   x := x || <head(z)>; z := tail(z)
   od
{x=F(x0,y0,z0,a0)}.
```

Here T corresponds to the last loop and again K is the identity function, CLOSURE holds and hence K itself is a reduction hypothesis. The proof of this program thus consists of showing ITERATION', i.e.

a=NULL, y≠NULL -> F(x,y,z,a)=F(x||<head(y)>,tail(y),z,a)

and then verifying

```
{a=NULL & y=NULL}
while z ≠ NULL do
   x := x || <head(z)>; z := tail(z)
   od
{x=F(x0,y0,z0,a0)}.
```

Again using the identity function as a reduction hypothesis, this remaining verification problem can be proved by showing ITERATION' and BOUNDARY', i.e.

a=NULL, y=NULL, z≠NULL ->
F(x,y,z,a)=F(x||<head(z)>,y,tail(z),a)

and

a=NULL, y=NULL, z=NULL -> x=F(x,y,z,a).

We remark that none of the program-oriented reduction hypothesis techniques defined in Theorem 3.2 assume any knowledge of the characteristics of T. Thus these techniques can be used for creating a reduction hypothesis in the circumstance where the input/output behavior of this subprogram is not known. The specification-oriented heuristic of Theorem 3.3, however, cannot be employed without this knowledge.

In Section 2.4, it was explained that the ease with which a composite program is verified depends largely on the strategy used to decompose the program and prove its correctness based on analysis of its components. While it is difficult to define an "optimal" decomposition strategy, we offer our view that the notion of a sequence of proof transformations presented in this section seems to represent a quite effective decomposition technique in practice. The proof proceeds in a "top down" fashion, decomposing the original verification problem into new, simpler problems. By way of contrast, the functional approach based on prime program decomposition is a more "bottom up" technique; each loop is analyzed and verified individually, without consideration of how the loop fits into the program as a whole. As a result, a functional proof based on prime program decomposition of the program is Example 3.7 would have required knowing the intended function of each of the three WHILE loops. This information is unnecessary when the program is verified by the sequence of proof transformations.

3.4. Discussion

From a practical point of view, it is difficult to carefully assess the relative merits of the proposed program-verification methodology. Preferences by people involved in verifying programs are often based on which methodology appears to be more "natural" or "intuitive" in a given application. Furthermore, answers to questions such as these no doubt are largely influenced by the way a person was trained in the field of software engineering.

Despite this caveat, we offer our view that in a number of cases, it seems "easier" to create a reduction hypothesis than it does to create an adequate inductive assertion or tail function for a proof by the standard techniques discussed in Section 3.2. Indeed, it is difficult to argue the reverse case in light of the results of that section, which state that any adequate inductive assertion is a reduction hypothesis, and that any tail function can be simply transformed to one. On the other hand, our feeling is that the verification conditions which result from the use of a reduction hypothesis seem somewhat more complicated than their counterparts in either of these standard techniques. This is largely due to the necessity of having three distinct data states appearing in condition ITERATION′ and two distinct data states appearing in condition BOUNDARY′. Both of these verification conditions, however, are usually easily simplified in practice.

In light of these comments, an interesting direction for future research is the translation of heuristics such as those defined in Theorems 3.2 and 3.3 into the framework of standard correctness techniques. For example, what, if any, is the impact of these correctness/incorrectness preserving transformations on the synthesis of an inductive assertion or tail function for the program under consideration? Some results along this line are presented in Chapter 4. In that chapter, heuristic SHORTCUT is applied to the problem of synthesizing tail functions for initialized loop programs and meets with a fair degree of success.

The solution of any complex problem is often best decomposed into solutions of appropriate simpler problems. This is an important principle on which our verification strategy is based. A search for a reduction hypothesis is really a search for a suitable simpler problem to substitute for the original. As discussed in Section 3.3, proving condition ITERATION′ for this simpler problem decomposes the solution of the new problem into the solution of a still simpler problem and so on.

4. A Heuristic For Deriving Loop Functions

In this chapter, we will consider programs of the following form:

```
<INITIALIZATION STATEMENTS>
while <LOOP PREDICATE> do
   <LOOP BODY STATEMENTS>
   od.
```

These programs tend to occur frequently in programming in order to accomplish some specific task, e.g. sort a table, traverse a data structure, calculate some arithmetic function, etc. More precisely, the intended purpose of such a program is often to compute, in some particular output variable(s), a specific function of the program inputs. In this chapter, we address the problem of analyzing a program of the above form in order to prove its correctness relative to this intended function.

One common strategy taken to solve this problem is to heuristically synthesize a sufficiently strong inductive assertion (i.e. loop invariant [Hoare 69]) for proving the correctness of the program. A large number of techniques to aid in the discovery of these assertions have appeared in the literature (see, for example, [Wegbreit 74, Katz & Manna 76]). It is our view, however, that these techniques seem to be more "machine oriented" than "people oriented." That is, they seem geared toward use in an assertion generator for an automatic program verification system. Furthermore, a sizable portion of the complexity of these techniques is due to their general-purpose nature. The methodology proposed here is intended to be used by programmers in the process of reading (i.e. understanding, documenting, verifying, etc.) programs and is tailored to the commonly occurring verification problem discussed above.

An alternative to the inductive assertion approach which is addressed in this chapter is to invent an hypothesis concerning the intended function (i.e. general input/output behavior) of the WHILE loop. Once this has been done, the loop can be proven/disproven correct with respect to the hypothesis using the functional technique described in Chapter 2 [Mills 72, Mills 75, Basu & Misra 75, Morris & Wegbreit 77, Wegbreit 77, Misra 78]. If the hypothesis is shown to be valid, the correctness/incorrectness of the program in question follows immediately. It has been shown [Basu & Misra 76, Misra 78, Misra 79, Basu 80] that this loop hypothesis can be generated in a deterministic manner (i.e. one that is guaranteed to succeed) for two restricted classes of programs. The approach suggested here is similar to this method in that the same type of loop behavior seems to be exploited in order to obtain the hypothesis. Our approach is not deterministic in general, but as a result, is intended to be more widely applicable and easier to use than those previously proposed in the literature.

One view of the problem of discovering the general input/output behavior of the WHILE loop under consideration might be to study it and make a guess about what it does. One might go about doing this by "executing" the loop by hand on several sample inputs and then guessing some general expression for the input/output behavior of the loop based on these results. Decisions that need to be made when using such a technique include how many sample inputs to use, how should these inputs be selected, and how should the general expression be inferred. Another consideration is that hand execution can be a difficult and an error prone task. Indeed, it seems that the loops for which hand execution can be carried out in a straightforward manner are the ones that are least in need of verification or some other type of formal analysis.

Our methodology is similar to this technique in that we attempt to infer the general behavior of the loop from several sample loop behaviors. In contrast to this technique, however, the sample behaviors are not obtained from hand execution, rather they are obtained from the specification for the initialized loop program. In many of the cases we have studied, the general behavior of the loop in question is quite easy to guess from these samples. This is not to say that the loop computes a "simple" function of its inputs or that the loop necessarily operates in a "simple" manner. Much more accurately, the ease with which the general behavior can be inferred from the samples is due to a "simple" connection between a change in the input value of an initialized variable and the corresponding change caused in the result produced by the loop. We will expand on this idea in what follows.

4.1. The Technique

In order to describe the proposed technique, we represent the verification problem discussed above as follows:

```
{X ∈ D(f)}
X := K(X);
while B(X) do
   X := H(X)
   od
{v=f(X0)}.
```

In this notation, X represents the data state of the program. K and H are data-state-to-data-state functions corresponding to the effects of the initialization and loop body respectively. B is a predicate over the data state. The program is specified to produce in the variable v of X a function f of the input data state X0.

If D is the set of all possible program data states and T is the set of values that the variable v may assume, the specification function f has the functionality f : D -> T. In order to

verify a program of this form, we choose to find a function g : D -> T which describes the input/output characteristics of the WHILE loop over a suitably general input domain. Specifically, this input domain must be large enough to contain all the intermediate data states generated as the loop iterates. If this is the case, the loop is said to be closed (see Section 2.1) for the domain of g.

We briefly consider two alternative approaches to synthesizing this loop function g. The alternatives correspond to the "top down" and "bottom up" approaches to creating inductive assertions discussed in [Katz & Manna 73, Ellozy 81]. In the "top down" alternative, the hypothesis g answers the question "what would the general behavior of the loop have to be in order for the program to be correct?" If such an hypothesis can be found and verified, the correctness of the program is established. If the program is incorrect, no such valid hypothesis exists. In the "bottom up" alternative, the hypothesis g answers the question "what is the general behavior of the loop?" In this case, a valid hypothesis always exists. Once it has been found and verified, the program is correct if and only if the initialization followed by g is equivalent to the function f.

The advantage of a "top down" approach is that it is usually easier to apply in practice because the verifier has more information to work with when synthesizing the hypothesis. The disadvantage of such an approach is that it may not be as well-suited to disproving the correctness of programs. This is because to disprove a program, the verifier must employ an argument which shows that there does not exist a valid hypothesis. The method described in this chapter is based on the "top down" approach. We will return to a discussion of this advantage and disadvantage later.

We begin by assuming the program in question is correct with respect to its specification. We then consider several properties of the function g which result from this assumption. First, the correctness of the program implies

(4.1) $X0 \in D(f) \rightarrow f(X0)=g(K(X0))$.

That is, for inputs satisfying the program precondition, the initialization followed by the loop yields the desired result. Secondly, since the loop computes g,

$$B(X0) \rightarrow g(X0)=g(H(X0))$$

holds by the "iteration condition" [Misra 78] of the standard technique for showing the loop computes g. This implies

$$B(K(X0)) \rightarrow g(K(X0))=g(H(K(X0))).$$

Combining with (4.1) yields

(4.2) $X0 \in D(f), B(K(X0)) \rightarrow f(X0)=g(H(K(X0)))$.

At this point we choose to introduce an additional universally quantified state variable X into each of (4.1) and (4.2), resulting in the equivalent conditions

(4.1') $X0 \in D(f), X=K(X0) \rightarrow g(X)=f(X0)$

and

(4.2') X0 ∈ D(f), B(K(X0)), X=H(K(X0)) -> g(X)=f(X0).
We summarize by saying that if the program is correct with respect to its specification, conditions (4.1') and (4.2') hold.

Suppose now that the specification (f), and the input/output behavior of the initialization (K), loop predicate (B) and loop body (H) are known. Given this, (4.1') and (4.2') can be used to solve for the loop hypothesis g on a certain set of inputs assuming the correctness of the program. Indeed, (4.1') and (4.2') can be thought of as defining portions of the unknown loop function g we are seeking. Specifically, each of (4.1') and (4.2') can be viewed as defining a function g with a restricted domain. In this light, for example, (4.1') defines the function (i.e. set of ordered pairs)

g = {(X,Z) | ∀ X0 ∈ D(f) ⊢ (X=K(X0) & Z=f(X0))}.

We call (4.1') and (4.2') constraint functions since they serve as constraints (i.e. requirements) on the general loop function. More precisely put, the constraint functions are subsets of the general loop function. The hope is that if these subsets are representative of the whole, the general loop function may be inferred through analysis of the constraint functions.

In what follows we describe a four step process for constructing a general loop function g from these constraint functions. We suggest that the reader not be taken aback by what may appear to be considerable complexity in the description of our technique. We intentionally have attempted to describe the procedure in a careful, precise manner. Furthermore, the technique is based on a few simple ideas and, once those ideas have been learned, we feel it can be applied with a considerable amount of success.

Example 4.1 - As we describe these steps, we will illustrate their application on the following trivial program to compute multiplication:

```
{v>=0}
z := 0;
while v ≠ 0 do
   z := z + k;
   v := v - 1
   od
{z=v0*k}.
```

We now proceed with a description of these steps.

Step 1 : RECORD - The first step consists of recording the constraint functions (copied from (4.1') and (4.2'))

C1: X0 ∈ D(f), X=K(X0) -> g(X)=f(X0)

and

C2: X0 ∈ D(f), B(K(X0)), X=H(K(X0)) -> g(X)=f(X0).

As a notational convenience, we dispense with the data-state notation and use program variables (possibly subscripted by 0 to

denote their initial values) in these function definitions. The terms X0 ∈ D(f) and f(X0) come from the pre- and post-condition for the initialized loop respectively. The term X=K(X0) is based on the input/output behavior of the initialization, and the terms B(K(X0)) and X=H(K(X0)) together describe the input/output behavior of the initialization followed by exactly one loop iteration. We illustrate these ideas with the multiplication program in Example 4.1. The constraint functions for this program are as follows:

C1: v0>=0, v=v0, z=0 -> g(z,v,k)=v0*k
C2: v0>0, v=v0-1, z=k -> g(z,v,k)=v0*k.

We make the following comments concerning these function definitions. First, in the interest of simplicity, we do not RECORD the "effect" of the initialization or loop body on the constant k (i.e. we dispense with k=k0 and the need for a symbol k0). Secondly, g is defined as a function of each program variable which occurs in the loop predicate or loop body. That is, g is a function of the variables on which the behavior of the loop directly depends. Furthermore, note that in C2, the term v0>0 captures both X0 ∈ D(f) (i.e. v0>=0) and B(K(X)) (i.e. v0≠0). As a final remark, in a constraint function we will use the phrase domain requirement to refer to the collection of terms to the left of the "->" symbol and function expression to refer to the expression which defines the value of g (e.g. v0*k in both C1 and C2 above).

Step 2 : SIMPLIFY - All variables which appear in the function definition but not in the argument list for g must eventually be eliminated from the definition. On occasion, it is possible to solve for the value of such a variable in the domain requirement and substitute the equivalent expression for it throughout the definition. To illustrate, in the definition C1 above, v0 is a candidate for elimination. We know its value as a function of v (i.e. v0=v), hence we can SIMPLIFY this definition to

C1: v>=0, z=0 -> g(z,v,k)=v*k.

Note that the term v=v0 has disappeared since with the substitution it is equivalent to TRUE. In a similar manner, the second constraint function can be SIMPLIFIED to (using v0=v+1)

C2: v>=0, z=k -> g(z,v,k)=(v+1)*k.

Although applying this simplifying heuristic is most often a straightforward process, care must be taken to insure that the domain of the constraint function is not mistakenly extended. For example, if d and d0 are integer variables, the definition

d0>0, d=d0*2 -> g(d)=d0*8

does not SIMPLIFY to

d>0 -> g(d)=d*4

since the first function defines a value of g only for positive, even values of d while the second definition defines a value of g for all positive d. The first function does SIMPLIFY to

d>0, EVEN(d) -> g(d)=d*4

where EVEN(d) is a predicate which is TRUE iff d is even.

Step 3 : REWRITE - Variables which appear in the argument list for g but not in the function expression of its definition are candidates to be introduced into the function expression. Each of these variables will be bound to a term in the domain requirement of the definition. The purpose of this step is to rewrite the function expression of C2 (based on the properties of the operation(s) involved) in order to introduce these terms into the function expression. To illustrate, consider the above SIMPLIFIED C2 definition. The variable z is a candidate to be introduced into the function expression (v+1)*k. It is bound to the term k in the domain requirement. Thus we need to introduce an additional term k into this function expression. One way to do this is to translate the expression to v*k+k. Based on this, we REWRITE C2 as

C2: v>=0, z=k -> g(z,v,k)=v*k+k.

Step 4 : SUBSTITUTE - In steps 2 and 3, the constraint functions are massaged into equivalent definitions in order to facilitate step 4. The purpose of this step is to attempt to infer a general loop function from these constraints. We motivate the process as follows. Suppose we are searching for a particular relationship between several quantities, say E, m and c. Furthermore, suppose that through some form of analysis we have determined that when m has the value 17, the relationship E=17*(c**2) holds. A reasonable guess, then, for a general relationship between E, m and c would be E=m*(c**2). This would be particularly true if we had reason to suspect that there was a relatively simple connection between the quantities m and E. We arrived at the general relationship by substituting the quantity m for 17 in the relationship which is known to hold when m has the value 17. Viewed in this light, the purpose of the constraint function C2 is to obtain a relationship which holds for a specific value of m (e.g. 17). The step REWRITE exposes the term 17 in this relationship. Finally, SUBSTITUTE substitutes m for 17 in the relationship and proposes the result as a general relationship between E, m and c. In terms of the multiplication program being considered, the SUBSTITUTE step calls for replacing one of the terms k in the above rewritten function expression with the term z. The two possible substitutions lead to the following general functions:

v>=0 -> g(z,v,k)=v*k+z

and

v>=0 -> g(z,v,k)=v*z+k.

Both of these (necessarily) are generalizations (i.e. supersets) of C2, however, only the first is also a generalization of C1. Hence this function is hypothesized as a description of the general behavior of the above WHILE loop.

We have applied the above 4 steps to obtain an hypothesis for the behavior of the loop in question. Since this description is sufficiently general (specifically, since the loop is closed for the domain of the function), we can prove/disprove the correctness of the hypothesis using standard verification

techniques [Mills 75, Misra 78]. Specifically, the hypothesis is valid if and only if each of

- the loop terminates for all v>=0,
- v=0 -> z=z + v*k, and
- z + v*k is a loop constant (i.e. v0*k0=z + v*k is a loop invariant)

hold. We remark that the loop hypothesis is selected in such a way that if it holds (i.e. the loop does compute this general function), the initialized loop is necessarily correct with respect to f.

We emphasize that there are usually an infinite number of generalizations of the constraint functions C1 and C2, and that, depending on how REWRITE and SUBSTITUTE are applied, the technique is capable of generating any one of these generalizations. For example, REWRITE and SUBSTITUTE applied to the multiplication example could have produced

```
C2:  v>=0, z=k -> g(z,v,k)=
     v*k + 3*k + k*k*(v-7)/(4*k) + k*k*k/(k*k)
               - k*k*k*(v-7)/(4*k*k) - k*k*k*3/(k*k)
```

and

```
     v>=0 -> g(z,v,k)=
     v*k + 3*z + z*z*(v-7)/(4*k) + z*z*z/(k*k)
               - z*z*z*(v-7)/(4*k*k) - z*z*z*3/(k*k)
```

respectively, where "/" denotes an integer division (with truncation) infix operator which yields 0 when its denominator is 0. This last function is also a generalization of C1 and C2.

It has been our experience, however, that many initialized loops occur in which there exists some relatively simple connection between different input values of the variables constrained by initialization and the corresponding result produced by the WHILE loop. Most often in practice, these variables are bound to values in the domain requirement of C2 which suggest an application of REWRITE that uncovers this relationship and leads to a correct hypothesis concerning the general loop behavior. In the following section we illustrate a number of example applications of this technique.

4.2. Applications

Example 4.2 - The following program computes integer exponentiation. This example serves to illustrate the use of the technique when the loop body contains several paths:

```
{d>=0}
w:=1;
while d ≠ 0 do
   if odd(d) then w := w * c fi;
   c := c*c; d := d/2
   od
{w=c0 ^ d0}.
```

The infix operator ^ appearing in the postcondition represents integer exponentiation. The first constraint function is easily RECORDED:

d0>=0, c=c0, d=d0, w=1 -> g(w,c,d)=c0^d0

and SIMPLIFIES to

C1: d>=0, w=1 -> g(w,c,d)=c^d.

Since there exist two paths through the loop body, we will obtain two second constraint functions. The first of these deals with the path which updates the value of w and is executed when the input value of d is odd. The function is

d0>0, odd(d0), w=c0, c=c0*c0, d=d0/2 -> g(w,c,d)=c0^d0

which SIMPLIFIES to

C2a: d>=0, c=w*w -> g(w,c,d)=w^(d*2+1).

The function corresponding to the other loop-body path is

d0>0, ~odd(d0), w=1, c=c0*c0, d=d0/2 -> g(w,c,d)=c0^d0

and SIMPLIFIES to

d>=0, w=1, SQUARE(c) -> g(w,c,d)=SQRT(c)^(d*2)

i.e.

C2b: d>=0, w=1, SQUARE(c) -> g(w,c,d)=c^d

where SQUARE(x) is a predicate which is TRUE iff x is a perfect square and SQRT(x) is the square root of the perfect square x. The term SQUARE(x) is necessary in the domain requirement since the unSIMPLIFIED function is only defined for values of c which are perfect squares. Note that C2b is a subset of C1 and hence is of no additional help in characterizing the general loop function. The heuristic suggested in REWRITE is to rewrite the function expression w^(d*2+1) of C2a in terms of w, w*w (so as to introduce c) and d. The peculiar nature of the exponent in this expression leads one to the equivalent formula w*((w*w)^d). Applying SUBSTITUTE in C2a yields

d>=0 -> g(w,c,d)=w*(c^d).

This function is in agreement with (i.e. is a superset of) C1 and thus is a reasonable hypothesis for the general loop function.

In this example, the portion of C2 corresponding to the loop-body path which bypasses the updating of the initialized data is a subset of C1. Based on this, one might conclude that such loop-body paths should be ignored when constructing C2. Considering all loop-body paths, however, does increase the likelihood that an incorrect program could be disproved (at the time the general loop function is being constructed) by observing an inconsistency between constraint functions C1 and C2. For instance, in the example, if the assignment to c had been written "c:=c*2", the above analysis would have detected an inconsistency in the constraints on the general loop function. Such an inconsistency implies that the hypothesis being sought for the general behavior of the loop does not exist, and hence, that the program is not correct with respect to its specification.

In the previous section, the reader may recall that awkwardness in disproving programs was offered as a disadvantage of a "top down" approach to synthesizing g. However, it has been our experience that, as in the above instance, an error in the

program being considered often manifests itself as an inconsistency between C1 and C2. Such an inconsistency is usually "easy" to detect and hence the program is "easy" to disprove. While it is difficult to give a precise characterization of when this will occur, intuitively, it will be the case provided that the "error" (e.g. c*2 for c*c) can be "executed" on the first iteration of the loop.

Example 4.3 - The following program counts the number of nodes in a nonempty binary tree using a set variable s. It differs from the previous example in that more than one variable is initialized. The tree variable t is the input tree whose nodes are to be counted. We use the notation left(t) and right(t) for the left and right subtrees of t respectively. The predicate empty(t) is TRUE iff t is the empty tree (i.e. contains 0 nodes).

```
{~empty(t)}
n := 0; s := {t};
while s ≠ {} do
   select and remove some element e from s;
   n := n + 1;
   if ~empty( left(e)) then s := s U {left (e)} fi;
   if ~empty(right(e)) then s := s U {right(e)} fi
   od
{n=NODES(t)}
```

The notation NODES(t) appearing in the postcondition stands for the number of nodes in binary tree t. The first constraint function is

C1: ~empty(t), n=0, s={} -> g(n,s)=NODES(t).

Rather than considering each of the four possible paths through the loop body individually, we abstract the combined effect of the two IF statements as the assignment

s := s U SONS(e),

where SONS(x) is the set of 0, 1 or 2 nonempty subtrees of x. Applying this, the second constraint function is

C2: ~empty(t), n=1, s=SONS(t) -> g(n,s)=NODES(t).

We choose to REWRITE the function expression for C2 using the recursive definition that NODES(x) for a nonempty tree x is 1 plus the NODES value of each of the 0, 1 or 2 nonempty subtrees of x. Specifically, this would be

1+SUM(x,SONS(t),NODES(x))

where SUM(A,B,C) stands for the summation of C over all A ∈ B. Applying SUBSTITUTE in the obvious way yields

~empty(t) -> g(n,s)=n+SUM(x,s,NODES(x))

which is in agreement with C1 and is thus a reasonable guess for the general loop function g.

Two remarks are in order concerning this example. The first deals with the condition ~empty(t) appearing in the domain requirement of the obtained function. The reader may wonder, if t is not referenced in the loop (it is not in the argument list

for g), how can the loop behavior depend on empty(t)? The answer is that it obviously cannot; the above function is simply equivalent to

g(n,s)=n+SUM(x,s,NODES(x)).

For the remainder of the examples of this section, we assume that these unnecessary conditions are removed from the domain requirement of the constraint function as part of the SUBSTITUTE step.

As a second point, in Example 4.3 we encounter the case where the obtained function is, strictly speaking, too general, in that its domain includes "unusual" inputs for which the behavior of the loop does not agree with the function. For instance, in the example, the loop computes the function

g(n,s)=n+SUM(x,s,NODES(x))

only under the provision that the set s does not contain the empty tree. This is normally not a serious problem in practice. One proceeds as before, i.e. attempts to push through a proof of correctness using the inferred function. If the proof is successful, the program has been verified; otherwise, the characteristics of the input data which cause the verification condition(s) to fail (e.g. s contains an empty tree) suggest an appropriate restriction of the input domain (e.g. s contains only nonempty trees) and the program can then be verified using this new, restricted function.

Example 4.4 [Gries 79] - Ackermann's function A(m,n) can be defined as follows for all natural numbers m and n:

```
A(0,n)     = n+1
A(m+1,0)   = A(m,1)
A(m+1,n+1) = A(m,A(m+1,n)).
```

The following program computes Ackermann's function using a sequence variable s of natural numbers. The notation s(1) is the rightmost element of s and s(2) is the second rightmost, etc. The sequence s(..3) is s with s(2) and s(1) removed. We will use < and > to construct sequences, i.e. a sequence s consisting of n elements will be written <s(n), ... ,s(2),s(1)>.

```
{m>=0,n>=0}
s := <m,n>;
while size(s) ≠ 1 do
   if s(2) = 0 then     s:=s(..3) || <s(1)+1>
   elseif s(1)=0 then   s:=s(..3) || <s(2)-1,1>
   else                 s:=s(..3) || <s(2)-1,s(2),s(1)-1> fi
   od
{s=<A(m,n)>}
```

For this program, the first constraint function is

C1: m>=0, n>=0, s=<m,n> -> g(s)=<A(m,n)>.

The second constraint functions corresponding to the 3 paths through the loop body are

C2a: m=0, n>=0, s=<n+1> -> g(s)=<A(m,n)>

C2b: m>0, n =0, s=<m-1,1> -> g(s)=<A(m,n)>
C2c: m>0, n >0, s=<m-1,m,n-1> -> g(s)=<A(m,n)>.
REWRITING these 3 based on the above definition of A yields
m=0, n>=0, s=<n+1> -> g(s)=<n+1>
m>0, n =0, s=<m-1,1> -> g(s)=<A(m-1,1)>
m>0, n >0, s=<m-1,m,n-1> -> g(s)=<A(m-1,A(m,n-1))>.
SUBSTITUTING here yields
s=<s(1)> -> g(s)=<s(1)>
s=<s(2),s(1)> -> g(s)=<A(s(2),s(1))>
s=<s(3),s(2),s(1)> -> g(s)=<A(s(3),A(s(2),s(1)))>.
Note that the second of these functions implies C1. The 3 seem to suggest the general loop behavior (where n>1)
g(<s(n),s(n-1), ... ,s(1)>) =
<A(s(n),A(s(n-1), ... A(s(2),s(1)) ...)>.

We remark that in the first 3 examples, the heuristic resulted in a loop function which was sufficiently general (i.e. the loop was closed for the domain of the inferred function). Example 4.4 illustrates that this does not always occur. The loop function heuristic is helpful in the example in that it suggests a behavior of the loop for general sequences of length 1, 2 and 3. Based on these results, verifier is left to infer a behavior for a sequence of arbitrary length.

Example 4.5 - Let v be a one dimensional array of length n>0 which contains natural numbers. The following program finds the maximum element in the array:

```
m := 0; i := 1;
while i <= n do
   if m < v[i] then m := v[i] fi;
   i := i + 1
   fi
{m=BIGGEST(v)}
```

The notation BIGGEST(v) appearing in the postcondition stands for the largest element of v. The following constraint functions are obtained
C1: m=0, i=1 -> g(m,i,v,n)=BIGGEST(v)
C2: m=v[1], i=2 -> g(m,i,v,n)=BIGGEST(v).
Noticing the appearance of v[1] and 2 in C2, we REWRITE BIGGEST(v) in C2 as MAX(v[1],BIGGEST(v[2..n])), where MAX returns the largest of its two arguments, and v[2..n] is a notation for the subarray of v within the indicated bounds. The generalization which suggests itself,
g(m,i,v,n)=MAX(m,BIGGEST(v[i..n])),
agrees with C1.

Example 4.6 - If p is a pointer to a node in a binary tree, let POST(p) be the sequence of pointers which point to the nodes in a postorder traversal of the binary tree pointed to by p. The following program constructs POST(p) in a sequence variable vs using a stack variable stk. We use the notation l(p) and r(p)

for the pointers to the left and right subtrees of the tree pointed to by p. If p has the value NIL, POST(p) is the empty sequence. The variable rt points to the root of the input tree to be traversed.

```
p := rt; stk := EMPTY; vs := <>;
while ~(p=NIL & stk=EMPTY) do
   if p≠NIL then
      stk <= p /* push p onto stk */ ;
      p := l(p)
   else
      p <= stk /* pop stk */ ;
      vs := vs || <p>;
      p := r(p) fi
   od
{vs = POST(rt)}.
```

Up until now, we have attempted to infer a general loop function from two constraint functions. Of course, there is nothing special about the number two. In this example, the "connection" between the initialized variables and the function values is not clear from the first two constraint functions and it proves helpful to obtain a third constraint function. Functions C1 and C2 correspond to 0 and 1 loop-body executions, respectively. The third constraint function C3 will correspond to 2 loop-body executions. We will use the notation (e1, ... ,en) for a stack containing the elements e1, ... ,en from top to bottom. The constraint functions for this program are

```
C1:                    p=rt,        stk=EMPTY,       vs=<> ->
      g(p,stk,vs)=POST(rt)
C2:  rt≠NIL,           p=l(rt),     stk=(rt),        vs=<> ->
      g(p,stk,vs)=POST(rt)
C3a: rt≠NIL, l(rt)≠NIL, p=l(l(rt)), stk=(l(rt),rt), vs=<> ->
      g(p,stk,vs)=POST(rt)
C3b: rt≠NIL, l(rt)=NIL, p=r(rt),    stk=EMPTY,     vs=<rt> ->
      g(p,stk,vs)=POST(rt).
```

Note that there are two third constraint functions. C3a and C3b correspond to executions of the first and second loop-body paths (on the second iteration), respectively. There is only 1 second constraint function since only the first loop-body path can be executed on the first iteration. Using the recursive definition of POST, we REWRITE C2, C3a and C3b as follows:

```
C2´:  rt≠NIL,           p=l(rt),     stk=(rt),        vs=<> ->
      g(p,stk,vs)=POST(l(rt)) ||<rt>|| POST(r(rt))
C3a´: rt≠NIL, l(rt)≠NIL, p=l(l(rt)), stk=(l(rt),rt), vs=<> ->
      g(p,stk,vs)=POST(l(l(rt))) ||<l(rt)>|| POST(r(l(rt)))
                                 ||<rt>|| POST(r(rt))
C3b´: rt≠NIL, l(rt)=NIL, p=r(rt),    stk=EMPTY,     vs=<rt> ->
      g(p,stk,vs)=<rt> || POST(r(rt)).
```

Applying SUBSTITUTE to each of C2´, C3a´ and C3b´ suggests

```
stk=(e1),    vs=<> -> g(p,stk,vs)=POST(p) ||<e1>||POST(r(e1))
stk=(e1,e2), vs=<> -> g(p,stk,vs)=POST(p) ||<e1>||POST(r(e1))
                                          ||<e2>||POST(r(e2))
```

stk=EMPTY -> g(p,stk,vs)=vs||POST(p)

respectively. The first 2 of these functions imply the following behavior for an arbitrary stack where vs has the value <>:

stk=(el, ..., en), vs=<> -> g(p,stk,vs) =
POST(p) || (<el>|| POST(el) ||...||<en>|| POST(en))

and in combination with the last function, the general behavior

stk=(el, ..., en) -> g(p,stk,vs) =
vs || POST(p) || (<el>|| POST(el) ||...||<en>|| POST(en))

is suggested.

In this section we have illustrated the use of our technique on a number of example programs. The reader has seen that the success of the method hinges largely on the way REWRITE is performed. What guidelines can be used in deciding how to apply this step? The general rule given above is to identify the variables that need to be introduced into the expression and then to rewrite the expression using the terms to which these variables are bound. For instance in Example 4.3, NODES(t) was rewritten using the terms 1 and SONS(t). Beyond this rule, however, the reader may have noticed an additional similarity in the way REWRITE was applied in these examples. If f is the function or operation the initialized loop program is intended to compute, each REWRITE step involved decomposing an application of f in some way. In Example 4.1, for instance, a multiplication operation was decomposed into an addition and multiplication operation; in Example 4.3, a NODES operation was decomposed into a summation and a number of NODES operations; in Example 4.5, a BIGGEST operation was decomposed into a MAX and a BIGGEST operation. In Section 4.5 we will characterize this idea of decomposing the intended operation of the initialized loop program and discuss several implications of the characterization for the proposed technique.

In Example 4.6, we saw that the technique generalizes to the use of three (and indeed an arbitrary number of) constraint functions. We have seen that each of these functions defines a subset of the general loop function g being sought. If the constraint functions themselves are sufficiently general, it may be that the first several of these functions, taken collectively, constitute a complete description of g. We consider this situation in the following section.

4.3. Complete Constraints

The technique described above for obtaining a general loop function is "nondeterministic" in that the constraint functions do not precisely identify the desired function; rather they serve as a formal basis from which intelligent guesses can be made concerning the general behavior of the loop. Our belief is that it is often easy for a human being to fill in the remaining "pieces" of the loop function "picture" once this basis has been established.

There exist, however, circumstances when the constraints do constitute a complete description of an adequate loop function. Specifically, this description may be complete through the use of one, two or more of the constraint functions. The significance of these situations is that no guessing or "filling in the picture" is necessary; the program can be proven/disproven correct using the constraints as the general loop function. In this section we give a formal characterization of this circumstance.

Definition 4.1 - For some N > 0, an initialized loop is N-closed with respect to its specification f iff the union of the constraint functions C1,C2, ... ,CN is a function g such that the loop is closed for the domain of g. In this case, the constraints C1,C2, ... CN are complete.

Thus if a loop is N-closed for some N>0, the union of the first N constraint functions constitutes an adequate loop function for the loop under consideration. Intuitively, the value N is a measure of how quickly (in terms of the number of loop iterations) the variables constrained by initialization take on "general" values.

Example 4.7 - The following program

```
{b>=0}
a := a + 1;
while b > 0 do
    a := a + 1;
    b := b - 1
od
{a=a0 + b0 + 1}
```

is 1-closed since the first constraint function is
C1: b0>=0, a=a0+1, b=b0 -> g(a,b)=a0+b0+1
which SIMPLIFIES to
b>=0 -> g(a,b)=a+b
and the loop is closed for the domain of this function. Thus C1 by itself defines an adequate loop function.

Initialized loops which are 1-closed seem to occur rarely in practice. Somewhat more frequently, an initialized loop will be 2-closed. For these programs, the loop function synthesis technique described above (using 2 constraint functions) is deterministic.

Example 4.8a - Consider the program

```
sum := 0;
while seq ≠ EMPTY do
   sum := sum + head(seq);
   seq := tail(seq)
   od
{sum=SIGMA(seq0)}.
```

The notation SIGMA(seq0) appearing in the postcondition stands for the sum of the elements in the sequence seq0. The program is 2-closed since the second constraint function is

C2: seq0≠EMPTY, sum=head(seq0), seq=tail(seq0) ->
g(sum,seq)=SIGMA(seq0)

which SIMPLIFIES to

g(sum,seq)=sum+SIGMA(seq).

The loop is trivially closed for the domain of this function.

Example 4.8b - As a second illustration of a 2-closed initialized loop, the following program tests whether a particular key appears in an ordered binary tree.

```
success := FALSE;
while tree ≠ NULL & ~success do
   if     name(tree) = key then success := TRUE
   elseif name(tree) < key then tree := right(tree)
   else                         tree := left(tree) fi
   od
{success = IN(key,tree0)}
```

The notation IN(key,tree0) is a predicate which is true iff key occurs in ordered binary tree tree0. This program is also 2-closed. Note that the first constraint function

C1: success=FALSE, tree=tree0 ->
g(success,tree,key)=IN(key,tree0)

SIMPLIFIES to

success=FALSE -> g(success,tree,key)=IN(key,tree).

If we consider the first path through the loop body, the second constraint function is

C2: success=TRUE, tree0≠NIL, tree=tree0, key=name(tree) ->
g(success,tree,key)=IN(key,tree0)

which SIMPLIFIES to

success=TRUE, tree≠NIL, key=name(tree) ->
g(success,tree,key)=IN(key,tree).

Although the domain of the union of these two functions is somewhat restricted, i.e.

{<success,tree,key> |
((~success) OR (tree≠NIL & key=name(tree)))},

the loop is nevertheless closed for this domain and hence the initialized loop is 2-closed.

Example 4.8c - Consider the sequence of initialized loops P1,P2,P3 ... defined as follows for each I>0:

```
PI : {x>=0}
     x := x * I;
     while x > 0 do
        x := x - 1;
        y := y + k
     od
     {y=y0 + x0*I*k}.
```

For any I>0, the first I constraint functions for program PI are

```
C1: x0>=0,  x=x0*I,      y=y0          -> g(z,y,k)=y0+x0*I*k
C2: x0>=1,  x=x0*I-1,    y=y0+k        -> g(z,y,k)=y0+x0*I*k
 .
 .
 .
CI: x0>=I-1,x=x0*I-(I-1),y=y0+k*(I-1)  -> g(x,y,k)=y0+x0*I*k.
```

These SIMPLIFY to

```
     x>=0, MI(x)        -> g(x,y,k)=y+x*k
     x>=0, MI(x+1)      -> g(x,y,k)=y+x*k
      .
      .
      .
     x>=0, MI(x+(I-1)) -> g(z,y,k)=y+x*k
```

where MI is a predicate which is TRUE iff its argument is a multiple of I. Since the union of these is the function

```
     x>=0 -> g(x,y,k)=y+x*k,
```

and the loop is closed for the domain of this function, we conclude that for each I>0, program PI is I-closed.

For many initialized loops which seem to occur in practice, however, there does not exist an N such that they are N-closed with respect to their specifications. This means that no finite number of constraint functions will pinpoint the appropriate generalization exactly; i.e. when applying the above technique in these situations, some amount of inferring or guessing will always be necessary. A case in point is the integer multiplication program from Example 4.1. The constraint functions C1,C2,C3, ... define the general loop behavior for z=0, z=k, z=2*k, ... etc. The program cannot be N-closed for any N since with input v=N+1, the last value of z will be (N+1)*k which is not in the domain of any of these constraint functions.

As a final comment concerning N-closed initialized loops, it may be instructive to consider the following intuitive view of these programs. All 1-closed and 2-closed initialized loops share the characteristic that they are "forgetful," i.e. they soon lose track of how "long" they have been executing and lack the necessary data to recover this information. This is due to the fact that intermediate data states which occur after an arbitrary number of iterations are <u>indistinguishable</u> from data states which occur after zero (or one) loop iterations. To illustrate, consider the 2-closed initialized loop of Example 4.8a which sums the elements contained in a sequence. After some arbitrary number of iterations in an execution of this program, suppose we

stop it and inspect the values of the program variables sum and seq. Based on these values, what can we tell about the history of the execution? The answer is not too much; about all we can say is that if sum is not zero then we know we have previously executed at least 1 loop iteration, but the exact number of these iterations may be 1, 10 or 10000.

By way of contrast, again consider the integer multiplication program of Example 4.1, an initialized loop we know not to be N-closed for any N. Suppose we stop the program after an arbitrary number of iterations in its execution. Based on the values of the program variables z, v and k, what can we tell about the history of the execution? This information tells us a great deal; for example, we know the loop has iterated exactly z/k times and we can reconstruct each previous value of the variable z.

Initialized loops which have the information available to reconstruct their past have the potential to behave in a "tricky" manner. By "tricky" here, we mean performing in such a way that depends unexpectedly on the history of the execution of the loop (i.e. on the effect achieved by previous loop iterations). The result of this loop behavior would be a loop function which was "inconsistent" across all values of the loop inputs and which could only be inferred from the constraint functions with considerable difficulty. We consider this phenomenon more carefully in the following section; for now we emphasize that it is precisely the potential to behave in this unpleasant manner that is lacking in 1-closed and 2-closed initialized loops and which allows their general behavior to be described completely by the first one or two constraint functions.

4.4. 'Tricky' Programs

The above heuristic suggests inferring g from two subsets of that function, C1 and C2. Constraint function C2 is of particular importance since REWRITE and SUBSTITUTE are applied to this function and it, consequently, serves to guide the generalization process. C2 is based on the program specification f, the initialization and the input/output behavior of the loop body on its first execution. In any problem of inferring data concerning some population based on samples from that population, the accuracy of the results depends largely on how representative the samples are of the population as a whole. The degree to which the sample defined in C2 is is representative of the unknown general function we are seeking depends entirely on how representative the input/output behavior of the loop body on the first loop iteration is of the input/output behavior of the loop body on an arbitrary subsequent loop iteration.

To give the reader the general idea of what we have in mind, consider the program to count the nodes in a binary tree in Example 4.3. If the loop body did something peculiar when, for

example, the set s contained two nodes with the same parent node, or when n had the value 15, the behavior of the loop body on its first execution would not be representative of its general behavior. By "peculiar" here, we mean something that would not have been anticipated based solely on input/output observations of its initial execution. An application of our heuristic on programs of this nature would almost certainly fail since (apparently) vital information would be missing from C1 and C2.

Example 4.9 - Consider applying the technique to the following program which is an alternative implementation of the integer multiplication program presented in Example 4.1:

```
{v>=0}
z := 0;
while v ≠ 0 do
   if z=0 then       z := k
   elseif z=k then   z := z * 2 * v
   else              z := z - k fi;
   v := v - 1
   od
{z=v0*k}.
```

The constraint functions C1 and C2 are identical to those for the program in Example 4.1 and we have no reason to infer a different function g. Yet this function is not only an incorrect hypothesis, it does not even come close to describing the general behavior of the loop. The difficulty is that the behavior of the loop body on its first execution is in no way typical of its general behavior. This is due to the high dependence of the loop-body behavior on the input value of the initialized variable z.

We make the following remarks concerning programs of this nature. First, our experience indicates that they occur very rarely in practice. Secondly, because they tend to be quite difficult to analyze and understand, we consider them "tricky" or poorly structured programs. Thirdly, the question of whether the (input/output) behavior of the loop body on the first iteration is representative of its behavior on an arbitrary subsequent iteration is really a question of whether its behavior when the initialized variables have their initial values is representative of its behavior when the initialized variables have "arbitrary" values. Put still another way, the question is whether the loop body behaves in a "uniform" manner across the spectrum of possible values of the initialized data.

In practice, a consequence of a loop body exhibiting this uniform behavior is that there exists a simply expressed connection between different input values of the initialized data and the corresponding result produced by the WHILE loop. It is the existence of such a connection which motivates the SUBSTITUTE step above and which is thus a necessary precondition for a successful application of the technique. This explains its failure

in dealing with programs such as that in Example 4.9. We make no further mention of these "tricky" programs, and in the following section discuss an informal categorization of "reasonable" programs and consider its implications for our loop function synthesis technique.

4.5. BU and TD Loops

In this section, we discuss general characteristics of many commonly occurring iterative programs. These characteristics are used to suggest two categories of these programs. This categorization is of interest since the above heuristic for synthesizing loop functions is particularly useful when applied to initialized loops in one of these categories.

In solving any particular problem, it often makes sense to consider certain instances of the problem as being "easier" or "harder" to solve than other instances. For example, with the problem of sorting a table, the ease with which the sort can be performed may depend on the size of the table, i.e. an instance of the problem for a table containing N elements might be harder to solve than an instance of the problem for a table containing N-1 elements. Similarly, if the problem is multiplying natural numbers, a*b might be easier to solve than (a+1)*b. This notion of "easier" and "harder" instances of a problem is particularly apparent for problems with natural recursive solutions. These solutions solve complex instances in terms of less complex instances and hence support the idea of one problem instance being easier to solve than another.

For the purpose of this discussion, we divide the data modified by the initialized loop under consideration into two sections: the accumulating data and the control data. The accumulating data is the specified output variable(s) of the loop. The remaining modified data is the control data and often serves to "guide" the execution of the loop and determine the point at which the loop should terminate. Both the accumulating data and the control data are typically (but not always) constrained by initialization in front of the loop.

Example 4.10a - In the program

```
{n>=0}
z := 1; t := 0;
while t ≠ n do
   t := t + 1;
   z := z * t
   od
{z=n!}
```

the variable z is the specified output of the loop and is hence the accumulating data. The other modified variable, t, is used to control the termination of the loop and is the control data.

In many cases, the control data can be viewed as representing an instance (or perhaps several instances) of the problem being solved. As the loop executes and the control data changes, the control data represents different instances of this problem. To illustrate, we can think of the control data t in the previous example as a variable describing a particular instance of the factorial problem. As the loop executes, the variable t takes on the values 0, 1, ..., n, and these values can be thought to correspond to the problems 0!, 1!, ... n!.

Based on these informal observations, we characterize a BU (from the Bottom Upward) loop as one where the control data problem instances are generated in order of increasing complexity, beginning with a simple instance and ending with the input problem instance to be solved. In the execution of a BU loop, the control data can be viewed as representing the "work" that has been accomplished "so far." We consider the factorial program above to be a BU loop. At any point in time, the "work" so far accomplished is t! and t moves *from* 0 (a simple factorial instance) to n (the input factorial instance).

Conversely, we characterize a TD (from the Top Downward) loop as one where the control data problem instances are generated in order of decreasing complexity, beginning with the input problem instance and ending with a simple problem instance. In the execution of a TD loop, the control data can be viewed as representing the "work" that remains to be done.

Example 4.10b - We consider the following alternative implementation of factorial to be a TD loop:

```
{n>=0}
z := 1; t := n;
while t ≠ 0 do
   z := z * t;
   t := t - 1
   od
{z=n!}.
```

As before z and t are the accumulating and control data respectively. The variable t moves *from* n (the input factorial instance) and ends with 0 (a simple factorial instance). After any iteration, the product n*(n-1)* ... *(t+1) has been accumulated, leaving t! as the "work" that remains to be done.

Example 4.11 - As an additional illustration, consider the following three initialized loops which compute integer exponentiation:

```
A: {y>=0}                B: {y>=0}                C: {y>=0}
   w:=1; t:=0;              w:=1; t:=y;              w:=1; c:=x; t:=y;
   while t≠y do             while t≠0 do             while t≠0 do
      w := w*x;                w := w*x;                if odd(t) then
      t := t+1                 t := t-1                    w := w*c fi;
      od                       od                       c:=c*c; t:=t/2
   {w=x^y}                  {w=x^y}                     od
                                                     {w=x^y}
```

As before, the symbol ^ is used as an infix exponentiation operator. We consider program A to be a BU loop. The control data t moves from 0 to y and corresponds to the problem instances x^0, ..., x^y. On the other hand, B is TD since the control data t moves from y to 0 and corresponds to the problem instances x^y, ..., x^0. Program C (similar to that in Example 4.2) is slightly more difficult to analyze. The control data is the pair <c,t>. The pair is initialized to <x,y> and ends with the value <c´,0>, where c´ is some complex function of x and y. It seems reasonable to consider <c,t> as representing the problem c^t. Hence we conclude C is also TD. This conclusion also makes sense in light of the fact that C is really an optimized version of B which saves iterations by exploiting the binary decomposition of y.

The characterization of BU and TD loops described here is, of course, an informal one and depends largely on one´s interpretation of the meaning or purpose of the control data. We classified the above programs by using what we considered to be the most "natural" or intuitive interpretation; other interpretations are always possible. Occasionally, two different interpretations of the control data seem equally valid and hence the program may be considered as either BU or TD, depending on one´s point of view. For example, consider the following program which adds up the elements in a subarray between indices p1 and p2:

```
sum := 0; i := p1;
while t <= p2 do
   sum := sum + a[i];
   i := i + 1
   od
{sum=ASUM(a[p1..p2])}.
```

The notation ASUM(a[p1..p2]) appearing in the postcondition stands for the summation of the elements in the indicated subarray. The question which arises in attempting to classify this program is as follows: as the control data i moves through the values p1, p1+1, ..., p2, is it most appropriate to think of it as representing the problem instance which has been solved (i.e. ASUM(a[p1..i])) or as representing the problem instance which remains to be solved (i.e. ASUM(a[i..p2])). Both views seem equally intuitive, that is, the program seems to be as much BU as it is TD.

As a final example, we refer back to the program in Example 4.3 which counts the nodes in a binary tree. It is clear n and the set variable s are the accumulating and control data respectively. Initially, s contains the tree whose nodes are to be counted; when the program terminates s is empty. In between, s contains various subtrees of the original tree. It seems natural to view the set as containing progressively simpler and simpler instances of the NODES problem since the trees in s consist of fewer and fewer nodes as the loop executes. Thus we classify the program as a TD loop.

We have seen that the problem-solving method taken by a BU loop is one of approaching the general problem instance from some simple problem instance. Of course, this problem-solving method is reasonable only when there exists some technique whereby one is guaranteed to "run into" the general problem instance. Our view is that in many cases, such a convergence technique either does not exist or requires so much support that the BU approach is not practical. This appears to be particularly true for programs dealing with sophisticated data types (i.e. something other than integers) and for programs requiring a high degree of efficiency in their number of iterations.

To help see this point of view, again consider the NODES program of Example 4.3. Previously we argued that this was a TD program. What would a BU program which computed the same function look like? The following program skeleton suggests itself:

```
n := 0;  t1 := "an empty tree";
while t1 ≠ t do
   "add a node to t1 to make it look more like t";
   n := n + 1
   od
{n=NODES(t)}.
```

Here, the tree variable t1 is the control data and it represents the problem NODES(t1). The difficulty with this attempt at a program solution is the implementation of the modification of t1. Such a modification requires close inspection (i.e. a traversal) of t in order to move t1 toward t. In light of this, it seems more reasonable to count the nodes of t while it is being inspected and to dispense altogether with the variable t1.

As an illustration of another circumstance where the BU approach seems unreasonable, the reader is encouraged to imagine a BU implementation of integer exponentiation which operates as efficiently as the exponentiation program C from Example 4.11. Again, a program skeleton suggests itself:

```
{y>=0}
w := 1;  c := ?;  d := 0;
while <c,d> ≠ <x,y> do
   c := sqrt(c);
   if ? then
      d := d * 2 + 1; w := w * c
   else d := d * 2 fi
   od
{w=x^y}.
```

Here, we are attempting to move the control data <c,d> toward <x,y> as fast as we moved it away from <x,y> in TD program C. As with the BU NODES program, the problem here is how to complete the program so as to achieve the desired effect. Our conclusion concerning this program is that supplying an appropriate initial value for c and determining the proper loop-body path to be executed requires such complexity that this approach is not a feasible alternative to program C.

In this section we have suggested two informal categories of initialized loop programs. We offered the opinion that the approach taken in a BU program solution has rather limited applicability and that TD programs tend to occur more frequently in practice. We feel that this characterization is useful as a study of opposing problem-solving philosophies but our main source of motivation is to investigate the kinds of commonly occurring programs on which the loop function synthesis technique described above works well.

Consider applying this technique to a general TD program. In the second constraint function, the control data is bound to a value which represents a slightly less complex instance of the general problem being solved by the initialized loop. In practice, the appearance of this value in the constraint function suggests the problem decomposition being exploited by the programmer in order to achieve the program result. Applying this decomposition in REWRITE leads quite naturally to the desired general loop function.

Example 4.12 - Consider the TD factorial program from Example 4.10b. The second constraint function is

C2: n>0, z=n, t=n-1 -> g(z,t)=n!

The control data t being bound to n-1 suggests REWRITTING n! as n*(n-1)!. This leads to the correct general loop function. On the other hand, consider the second constraint function for the BU factorial program from Example 4.10a:

C2: n>0, z=1, t=1 -> g(z,t,n)=n!

How can the expression n! be rewritten in terms of 1, 1 and n? To obtain the correct general function, the expression would have to be rewritten as (1*n!)/(1!) which seems much less intuitive than that required for the TD version. As another point of comparison, consider the second constraint function for the TD exponentiation program B from Example 4.11:

C2: y>0, w=x, t=y-1 -> g(w,t,x)=x^y

and the second constraint function for the BU exponentiation program A from the same example:

C2: y>0, w=x, t=1 -> g(w,t,x,y)=x^y.

In both cases, the proper loop function may be obtained by using the REWRITE rule x^y = x*(x^(y-1)); however, this particular rule seems more strongly suggested in the constraint function for the TD program.

We remark that the same general phenomenon occurs with TD programs in the event the control data has been SIMPLIFIED out of the domain requirement for C2. In this case, the fact that the control data represents a slightly less complex instance of the general problem being solved manifests itself in the function expression for the SIMPLIFIED C2 being a slightly more complex instance of the problem being solved. For example, the constraint function C2 above for the TD exponentiation program B of Example 4.11 can be SIMPLIFIED to

t>=0, w=x -> g(w,t,x)=x^(t+1).

Before, the appearance of y-1 in the domain requirement suggested rewritting x^y as x*(x^(y-1)). Here, the appearance of t+1 in the function expression suggests rewritting x^(t+1) as x*(x^t) (see also Examples 4.1 and 4.2).

Suppose f is the operation or function the initialized loop program is intended to compute. In Section 4.2 we observed that each REWRITE in the examples of that section involved "decomposing" an application of f. This decomposition corresponds to rewritting that problem instance in terms of a slightly less complex problem instance (or instances). In general, of course, there are many ways this decomposition can be performed. In the examples of that section, however, as with all TD programs, the nature of the control data serves to guide this decomposition and thus tends to make the REWRITE step quite straightforward in practice.

The reader may have noticed that the general loop functions for the BU factorial and exponentiation programs contain more program variables and operations on those variables than their TD counterparts. For instance, the general loop functions for the BU and TD factorial programs are

0<=t<=n -> g(z,t,n)=z*(n!/t!)

and

0<=t -> g(z,t)=z*t!

respectively. This fact, by itself, helps explain why the loop function synthesis technique seems more difficult to apply on BU programs. It would be a mistake, however, to assume that the BU programs are more "complex" or are more difficult to analyze or prove. We consider TD loops to be somewhat more susceptible to the form of induction employed in functional loop verification. More precisely, the inductive hypothesis required in this type of proof (i.e. a general statement concerning the loop input/output behavior) seems to be more easily stated for TD programs than for

BU programs. On the other hand, BU programs seem somewhat more susceptible to an inductive assertion proof. The inductive hypothesis required in this type of proof (i.e. a sufficiently strong loop invariant) involves fewer program variables and operations on those variables than the same type of hypothesis for the corresponding TD loop. As an example the BU and TD factorial programs have adequate loop invariants $0<=t \& z=t!$ and $0<=t<=n \& z=n!/t!$ respectively.

In [Manna & Waldinger 70], the authors describe a program synthesis technique and point out that their method produces either of the above factorial programs depending upon which type induction rule the synthesizer is given to employ.

4.6. Related Work

In [Basu & Misra 76, Misra 78, Misra 79], the authors describe two classes of "naturally provable" programs for which generalized loop specifications can be obtained in a deterministic manner. The technique proposed in this chapter sacrifices determinism in favor of wide applicability and ease of use. It handles in a fairly straightforward manner typical programs in these two program classes (e.g. Examples 4.1-4.3) as well as a number of programs which do not fit in either of the classes (e.g. Examples 4.4-4.6).

Due to the close relationship between loop functions and loop invariants (as discussed, for example, in Chapter 2), any technique for synthesizing loop invariants can be viewed as a technique for synthesizing general loop functions (and vice versa). In this light, our method bears an interesting resemblance to a loop invariant synthesis technique described in [Wegbreit 74, Katz & Manna 76]. In this technique stronger and stronger "approximations" to an adequate loop invariant are made by pushing the previous approximation back through the loop once, twice, etc.

By way of illustration, consider the exponentiation program of Example 4.2. The loop exit condition can be used to obtain an initial loop invariant approximation

$d=0 \rightarrow w=c0^{d0}$.

This approximation can be strengthened by pushing it back through the loop to yield

$(d=0 \rightarrow w=c0^{d0}) \& (d=1 \rightarrow w*c=c0^{d0})$.

In the analysis presented in Example 4.2, we obtained a value for the generalized function specification for each of two different values of the initialized variable w (i.e. 1 and SQRT(c)); here we have obtained a "value" for the loop invariant we are seeking for each of two different values of the variable which controls the termination of the loop d. Applying the analysis in [Morris & Wegbreit 77], these loop invariant "values" can be translated to constraint functions as follows:

$d=0 \rightarrow g(w,c,d)=w$,

d=1 -> g(w,c,d)=w*c.

Of course, the function expression w*c in the second constraint can be rewritten w*(c^1); SUBSTITUTING as usual suggests the general loop function

g(w,c,d)=w*(c^d).

If we then add the program precondition as a domain restriction on this function, the result is the same general loop function discovered in Example 4.2.

We summarize the relationship between these two techniques as follows. As the initialized loop in question operates on some particular input, let X[0], X[1], . . ,X[N] be the sequence of states on which the loop predicate is evaluated (i.e. the loop body executes N-1 times). Of course, in X[0], the initialized variables have their initial values, and in X[N], the loop predicate evaluates to FALSE. The method proposed in this chapter suggests inferring the unknown loop function g from X[0], X[1], g(X[0]) and g(X[1]). The loop invariant technique described above, when viewed as a loop function technique, suggests inferring g from X[N], X[N-1], g(X[N]) and g(X[N-1]). Speaking roughly then, one technique uses the first several executions of the loop, the other uses the last several executions. One ignores the information that the loop must compute the identity function on inputs where the loop predicate is FALSE, the other ignores the information that the loop must compute like the initialized loop when initialized variables have their initial values.

Earlier we discussed "top down" and "bottom up" approaches to synthesizing g and indicated that our technique fit in the "top down" category. The technique based on the last several iterations is a "bottom up" approach. It is difficult to carefully state the relative merits of these two opposing techniques. In our view, however, there are a number of circumstances under which the technique based on the first several loop executions seems more "natural" and easily applied. These examples include the NODES program, the program to compute Ackermann's function and the TD factorial program discussed above. The reason is that a critical aspect of the general loop function is the function computed by the initialized loop program (e.g. exponentiation in the above illustration). In the technique based on the first several iterations, this function appears explicitly in the constraint functions. In the other technique, this information must somehow be inferred from the corresponding constraint functions (e.g. by looking for a pattern in these functions, etc.). This difficulty is inherent in any "bottom up" approach to synthesizing g.

In Chapter 3, we presented several guidelines for the synthesis of reduction hypotheses. Heuristic SHORTCUT of Theorem 3.2 in that chapter is based on the observation that the programs

A: P1;
<u>while</u> B1 <u>do</u>

B: P1; P2;
<u>while</u> B1 <u>do</u>

P2 P2
od od

are equivalent on inputs which require at least one loop iteration on program A. This is the underlying WHILE loop property that is used to obtain constraint function C2 in the loop function derivation technique described in this chapter. Specifically, C2 for the program A is exactly constraint function C1 for the program B.

4.7. Discussion

In this chapter we have proposed a technique for deriving functions which describe the general behavior of a loop which is preceded by initialization. These functions can be used in a functional [Mills 75] or subgoal induction [Morris & Wegbreit 77] proof of correctness of the initialized loop program. It is not our intention to imply that verification should occur after the programming process has been completed. There are, however, a large number of existing programs which must be read, understood, modified and verified by "maintenance" personnel. We offer the heuristic as a tool which is intended to facilitate these tasks.

It has been argued [Misra 78] that the notion of closure of a loop with respect to an input domain is fundamental in analyzing the loop. In Section 4.3, this idea is applied to initialized loop programs. The result is that a loop function g for a loop which is N-closed (for some N>0) can be synthesized in a deterministic manner by considering the first N constraint functions. Hence this categorization can be viewed as one measure of the "degree of difficulty" involved in verifying initialized loop programs.

An interesting direction for future research is the development of a precise characterization of programs which are not "tricky" (as discussed in Section 4.4). Some results along this line are described in Chapter 5 (see also [Basu 80]).

In Section 4.5 we discussed on an informal level the opposing BU and TD problem-solving strategies and their corresponding initialized loop realizations. We argued that the TD approach appeared to be more widely applicable and that, in practice, TD programs seem to occur more frequently. We explained the success of the proposed loop function creation technique on these programs in terms of an easily applied REWRITE step. These results are offered to help support our view that the technique may be employed successfully in a wide range of applications.

5. Analyzing Uniformly Implemented Loops

Consider the problem of proving/disproving a WHILE loop correct with respect to some functional specification f, i.e. f requires the output variable(s) to be some function of the inputs to the loop. If the loop precondition is weak enough so that the domain of f contains the intermediate states which appear after each loop iteration (i.e. if the loop precondition is a loop invariant), the loop is said to be closed for the domain of f. In Chapter 2 (Theorem 2.1) we saw that if the loop is closed for the domain of its specification, there are two easily constructed verification conditions based solely on the specification, loop predicate and loop body which are necessary and sufficient conditions for the correctness of the loop (assuming termination) with respect to its specification [Mills 75, Misra 78]. If the loop is not closed for the domain of the specification function, a generalized specification (i.e. one that implies the original specification) which satisfies the closure requirement must be discovered before these verification conditions can be constructed (this problem is analogous to that of discovering an adequate loop invariant for an inductive assertion proof [Hoare 69] of the program).

We remark that the restricted specification often occurs in the process of analyzing an initialized WHILE loop, i.e. one that consists of a WHILE loop preceded by some initialization code. This initialization typically takes the form of assignments of constant values to some of the variables manipulated by the loop. Examples include setting a counter to zero, a search flag to FALSE, a queue variable to some particular configuration, etc. It is clear that the initialized loop is correct with respect to some specification if and only if the WHILE loop by itself is correct with respect to a slightly modified specification. This specification has the same postcondition as the original specification and a precondition which is the original precondition together with the condition that the initialized variables have their initialized values. Since the initialized variables will typically assume other values as the loop iterates, the loop most likely will not be closed for the domain of this specification and a generalization of it will be necessary in order to verify the correctness of the program.

Example 5.1 - The following program multiplies natural numbers using repeated addition:

```
{v>=0,k>=0}
z := 0;
while v > 0 do
    z := z + k;
    v := v - 1
od
{z=v0*k}.
```

The term v0 appearing in the postcondition refers to the initial value of v. The program is correct if and only if

```
{z=0,v>=0,k>=0}
while v > 0 do
    z := z + k;
    v := v - 1
od
{z=v0*k}
```

is correct. Since this loop precondition requires z to have the value 0 and z assumes other values as the loop executes, the loop is not closed for this precondition. Thus, before this program can be verified using the above mentioned technique, this specification must be generalized to something like

```
{v>=0,k>=0}
while v > 0 do
    z := z + k;
    v := v - 1
od
{z=z0 + v0*k}
```

where z0 refers to the initial value of the variable z.

The approach to this problem suggested here is one of observing how particular changes in the value of some input variable (e.g. z in the example) affect the result produced by the loop body of the loop under consideration. Clearly in general, a change in the value of an input variable may cause an arbitrary (and seemingly unrelated) change in the loop-body result. In many commonly occurring cases, however, the result produced by the loop body is "uniform" across the entire spectrum of possible values for the input variable. It is this property that will be exploited in order to obtain a generalized specification for the loop being analyzed. The generalizations considered here have the property that the loop is correct with respect to the generalization if and only if the loop is correct with respect to the original specification. Thus if the loop is closed for the domain of the generalization, the program can be proven/disproven by testing its correctness relative to the generalization.

It is natural to expect that the ease with which a generalized specification may be obtained for a loop would depend largely on the nature of the loop. Results in [Wegbreit 77], for example, show that the problem of generalizing the loop specification for any program in a particular class of programs is NP-complete. On the other hand, work presented here and elsewhere [Basu & Misra 76, Misra 79, Basu 80], indicates that there do exist categories of loops for which generalized specifications can be obtained in a direct, routine manner. We feel that the notion of "uniform" loop-body behavior discussed in this chapter is valuable not only as a tool by which such generalizations may

be obtained, but also as an attempt at a characterization of loops which are susceptible to routine analysis, and hence in this sense, easy to verify and comprehend.

The following section defines the necessary notation and terminology and then introduces the idea of a generalized loop specification. Section 5.2 defines a uniformly implemented loop and states several implications of this definition for the problem of generalizing a specification for such a loop. These results are applied on several example programs in Section 5.3. In Section 5.4, a simplified procedure is suggested for proving/disproving a uniformly implemented loop correct with respect to the obtained generalization. Finally, several guidelines for recognizing uniformly implemented loops are presented in Section 5.5.

5.1. Preliminaries

We will consider a verification problem of the form

$\{\langle z,y\rangle \in D(f)\}$
while $B(\langle z,y\rangle)$ **do**
  $z,y := h'(z,y), h''(z,y)$
od
$\{\langle z,y\rangle = f(\langle z0,y0\rangle)\}$.

In this problem, f is a data-state-to-data-state function. The data state consists of two variables, z and y. The terms z0 and y0 refer to the initial values of z and y respectively. The effect of the loop body is partitioned into two functions h' and h'' which describe the new values of z and y respectively.

The loop will be referred to as P. As before, the data-state-to-data-state function computed by the loop (which, presumably, is not explicitly known) will be denoted [P]. Thus D([P]) is the set of states for which P terminates. As a shorthand notation we will use X for the state <z,y>, and H for the data-state-to-data-state function computed by the loop body, i.e.

$$H(X) = H(\langle z,y\rangle) = \langle h'(z,y), h''(z,y)\rangle.$$

Suppose the loop is not closed for D(f) in that this set contains only a restricted collection of values (maybe only one) of z and that other intermediate values of z occur as the loop iterates. The variable z will be called the key variable. Our goal here is to discover some more general specification f' which includes each of these intermediate values of the key variable in its domain. This generalization process (in one form or another) is necessary for a proof of correctness of the program under consideration.

Definition 5.1 - P is **correct** wrt a function f iff for all X in D(f), [P](X) is defined and [P](X)=f(X).

Definition 5.2 - A superset f' of f is a valid generalization of f iff if P is correct wrt f, then P is correct wrt f'.

Note that the collection of supersets of f is partially ordered by "is a valid generalization of." The following definition supplies the notation we will use to describe generalizations of the specification function f.

Definition 5.3 - If S is a set of ordered pairs of data states, f' is the extension of f defined by S iff f' is a function and f' is the union of f and S.

Definition 5.4 - If g is the extension of f defined by

(5.1) $\{(X,X) \mid \sim B(X)\}$,

then g is the base generalization of f.

Thus, if g is the base generalization of f, then

$g(X1)=X2 \iff (f(X1)=X2 \text{ OR } (\sim B(X1) \& X1=X2))$.

Throughout this chapter, we will continue to use the function symbol g for the base generalization of f. We remark that g exists provided the union of f and (5.1) is a function, i.e. provided

$X \in D(f) \& \sim B(X) \rightarrow f(X)=X$

holds. If this condition is not satisfied, P is not correct wrt f. Hence, if P is correct wrt f, the base generalization of f exists.

Theorem 5.1 - If g is the base generalization of f, then g is a valid generalization of f.

Proof - Suppose P is correct wrt f. We must show P is correct wrt g. Let $X \in D(g)$. If $X \in D(f)$, the loop handles the input correctly by hypothesis. If X is not in D(f), we must have $\sim B(X)$ and $g(X)=X$. Thus the program and g map X to itself and thus are in agreement. Consequently P is correct wrt g, and g is a valid generalization of f.

The theorem utilizes the fact that the loop must necessarily compute the identity function over inputs where the loop predicate is false. Combining this information with the program specification f results in a valid generalization of f.

Definition 5.5 - A valid generalization f' of f is adequate if the loop is closed for D(f').

The important characteristic of an adequate valid generalization f' is that it can be used to prove/disprove the correctness of P wrt the original specification f. Since the loop is closed for D(f'), P can be proven/disproven correct wrt f' using standard techniques [Mills 72, Mills 75, Basu & Misra 75, Morris & Wegbreit 77, Wegbreit 77, Misra 78]. Specifically, P is correct wrt f' iff each of

(5.2) the loop terminates for all $X \in D(f')$

(5.3) $X \in D(f') \;\&\; \neg B(X) \rightarrow f'(X)=X$
(5.4) $X \in D(f') \;\&\; B(X) \rightarrow f'(X)=f'(H(X))$
hold. If P is correct wrt f', then P is necessarily correct wrt any subset of f', including f. If P is not correct wrt f', then by the definition of a valid generalization, P must not be correct wrt f.

Example 5.2 - The following program tests whether a particular key appears in an ordered binary tree.

```
{success=FALSE}
while tree ≠ NULL & ~success do
   if     name(tree) = key then success := TRUE
   elseif name(tree) < key then tree := right(tree)
   else                         tree := left(tree) fi
   od
{success = IN(tree0,key)}
```

The function IN(tree0,key) appearing in the postcondition is a predicate which means "the ordered binary tree tree0 contains a node with name field key." The boolean variable success is chosen as the key variable since it is constrained to the value FALSE in the input specification. Thus success plays the role of z and the pair of variables <tree,key> correspond to y in the program schema discussed above. The specification function f is

f(<FALSE,tree,key>) = <IN(tree,key),tree',key'>

where tree' and key' are the final values of the variables tree and key computed by the loop, respectively. That is, since the final values of these variables are not of interest in this example, we specify these final values so as to be automatically correct. Using Theorem 5.1, a valid generalization of this specification is

```
g(<success,tree,key>) = if ~success then
                              <IN(tree,key),tree',key'>
                        else if tree=NULL OR success then
                              <success,tree,key>,
```

which is equivalent to

g(<success,tree,key>)=<success OR IN(tree,key),tree',key'>.

In this example, the domain of the base generalization g of f includes each value of the key variable, (i.e. FALSE and TRUE) and is thus adequate. Consequently, this generalization can be used to prove/disprove the correctness of the program.

In most cases, however, the heuristic suggested in Theorem 5.1 is insufficient to generate an adequate generalization. Indeed, the base generalization is an adequate generalization only in the case when the sole reason for the closure condition not holding is the existence of potential final values of the key variable (e.g. TRUE in the example) which are absent from D(f). In order to obtain a generalization that includes general values of the key variable, an important characteristic of the loop body which seems to be present in many commonly occurring loops will

be exploited.

5.2. Uniformly Implemented Loops

Definition 5.6 - Let P be a loop of the form described above. Let C be a set, let Z be the set of values the key variable z may assume, and let Y be the set of values the remaining variable(s) y may assume. Let

$$\$' : C \times Z \to Z$$

be an infix binary operator. The loop P is uniformly implemented wrt $' iff each of

(5.5) $B(\langle z,y\rangle) \to h'(c \mathbin{\$'} z,y) = c \mathbin{\$'} h'(z,y)$

(5.6) $B(\langle z,y\rangle) \to h''(c \mathbin{\$'} z,y) = h''(z,y)$

(5.7) $B(\langle z,y\rangle) \to B(\langle c \mathbin{\$'} z,y\rangle)$

hold for all $c \in C$, $z \in Z$ and all $y \in Y$.

Conditions (5.5) and (5.6) of this definition state that a modification to the key variable by the operation $' causes a slight but orderly change in the result produced by the loop body. The change is slight because the only difference in the result produced by the loop body occurs in the key variable. This difference is orderly because it corresponds precisely to the same $' operation that served to modify the input value of the key variable. Condition (5.7) specifies that such a modification does not cause the loop predicate B to change from true to false.

As a shorthand notation we define the infix operator $ as

$$c \mathbin{\$} X = c \mathbin{\$} \langle z,y\rangle = \langle c \mathbin{\$'} z,y\rangle.$$

In this notation (5.5)-(5.7) are equivalent to

(5.8) $B(X) \to c \mathbin{\$} H(X) = H(c \mathbin{\$} X)$

and

(5.9) $B(X) \to B(c \mathbin{\$} X)$.

Example 5.3 - Consider again the program from Example 5.1 which multiplies natural numbers using repeated addition:

```
{z=0,v>=0,k>=0}
while v>0 do
   z := z + k;
   v := v - 1
   od
{z=v0*k}.
```

Let z be the key variable. The pair <v,k> corresponds to the variable y occurring in the above schema. The loop is uniformly implemented wrt +, where C and Z are both the set of natural numbers. Note that adding some constant to the input value of z has the effect of adding the same constant to the value of z output by the loop body. Now consider the following alternative implementation of multiplication:

```
{z=0,v>=0,k>=0}
while v>0 do
      if k=0 & z=0 then z := v - 1
  elseif k=0 & z≠0 then z := z - 1
  elseif z<k       then z := z + k
  elseif z=k       then z := z * 2 * v
  else                 z := z - k fi;
  v := v - 1
  od
{z=v0*k}.
```

Again, let z be the key variable. This loop is not uniformly implemented wrt +. Intuitively, this is due to the high degree of dependence of the loop-body behavior on the value of the key variable. The result of this dependence is that adding some constant to the value of z causes an unorderly change in the value of z output by the loop body.

The reader may wonder if the second multiplication program above might be uniformly implemented wrt some operation other than +. We remark that any loop is uniformly implemented wrt $' : C x Z -> Z defined by

$$c \$' z = z$$

for all c ∈ C and z ∈ Z. For the purpose of our research, we rule out such trivial operations, i.e. we require that for some z ∈ Z, there exists some c ∈ C such that

$$c \$' z \neq z.$$

With this assumption, there does not exist an operation wrt which the second of the above loops is uniformly implemented (or more briefly, the loop is not uniformly implemented). To see this, suppose the loop were uniformly implemented wrt $' : C x Z -> Z. Let c0 and z0 be some fixed elements of C and Z, respectively, which satisfy

$$c0 \$' z0 \neq z0.$$

Since (5.5) must hold for all c ∈ C, z ∈ Z and all y ∈ Y, we choose z=z0, c=c0 and k=c0 $' z0. Applying (5.5) gives

$$v>0 \rightarrow h'(c0 \$' z0,\langle v,c0 \$' z0\rangle)=c0 \$' h'(z0,\langle v,c0 \$' z0\rangle)$$

for all v. We consider 3 exhaustive cases based on the values of c0 and z0. First, suppose c0 $' z0 = 0. Then we must have (since z0 ≠ 0)

$$v>0 \rightarrow v-1=c0 \$' (z0-1).$$

Since this must hold for all v and the value c0 $' (z0-1) is fixed by the original selection of c0 and z0, this is a contradiction. Next, suppose c0 $' z0 ≠ 0 and z0 < c0 $' z0. Then

$$v>0 \rightarrow (c0 \$' z0)*2*v=c0 \$' (z0+(c0 \$' z0))$$

for all v. Again, since the expression to the right of the equality sign is fixed and (c0 $' z0)*2*v varies with different values of v (c0 $' z0 is nonzero), this is a contradiction. The third case, where c0 $' z0 ≠ 0 and z0 > c0 $' z0, leads to a similar contradiction, and thus (5.5) does not hold. We conclude that the second multiplication program above is not uniformly implemented. That is, there does not exist a nontrivial modification that can be applied to the variable z which always results

in a slight and orderly change in the result produced by the loop body.

The results presented here are based on the following lemma concerning uniformly implemented loops. The lemma describes the output of a uniformly implemented loop P for some modified input c $ X (i.e. [P](c $ X)) in terms of the output of the loop for the input X (i.e. [P](X)) and the output of the loop for the input c $ [P](X) (i.e. [P](c $ [P](X))).

Lemma 5.1 - Let P be uniformly implemented wrt $'. Then
(5.10) X ∈ D([P]) -> [P](c $ X)=[P](c $ [P](X)).

Proof - We use induction on the number of iterations of P on X. For the base case of 0 iterations, [P](X)=X, and the lemma holds. Suppose it holds for all input data states X requiring n-1 iterations where n > 0. Let X1 require n iterations. Since n > 0, B(X1) holds. By (5.9), B(c $ X1). Note that H(X1) requires n-1 iterations on P; thus by the inductive hypothesis

[P](c $ H(X1)) = [P](c $ [P](H(X1))).

Due to the uniform implementation this is

[P](H(c $ X1)) = [P](c $ [P](H(X1))).

Using the loop property B(X) -> [P](X)=[P](H(X)) on both sides we get

[P](c $ X1) = [P](c $ [P](X1)).

Thus the inductive step holds and the lemma is proved.

The general idea behind our use of the lemma is as follows. Suppose the value [P](X) is known for some particular X. I.e. suppose we know what the loop produces for the input X. In addition, suppose that, given the result [P](X), the quantity [P](c $ [P](X)) is also known. With this information, we can then use Lemma 5.1 to "solve" for the (possibly unknown) value [P](c $ X). This additional information concerning the input/output behavior of the loop can be used as an aid in constructing a valid generalization of the specification f.

How can we find the value [P](X) and then the value [P](c $ [P](X)) for some X? The key lies in assuming the loop P is correct wrt f. If P is not correct wrt f, any generalization of f obtained by the technique will be a valid generalization by definition. Under this assumption, [P](X) is known for X ∈ D(f), i.e. X ∈ D(f) -> [P](X)=f(X), and hence Lemma 5.1 implies
(5.11) X ∈ D(f) -> [P](c $ X)=[P](c $ f(X)).

Consider now the base generalization g of f defined in the previous section. Recall that g is simply f augmented with the identity function over the domain where the loop predicate B is false. Assuming as before that P is correct wrt f, P is then correct wrt g by Theorem 5.1; hence X ∈ D(g) -> [P](X)=g(X). Thus (5.11) implies
(5.12) X ∈ D(f) & c $ f(X) ∈ D(g) -> [P](c $ X)=g(c $ f(X)).
Thus we can "solve" for the behavior of the loop on the input c $

X, assuming X ∈ D(f), c $ f(X) ∈ D(g) and P is correct wrt f. This suggests that if f′ is the extension of f defined by

(5.13) {(c $ X,g(c $ f(X))) | X ∈ D(f) & c $ f(X) ∈ D(g)},

then f′ is a valid generalization of f. Before giving a formal proof of this result, however, we first consider the question of the existence of such an extension of f. Specifically, it could be that for some c and X satisfying X ∈ D(f) and c $ f(X) ∈ D(g), that c $ X ∈ D(f) and f(c $ X) ≠ g(c $ f(X)). This would imply that the extension of f defined by (5.13) does not exist. The following theorem states that this implies P is not correct wrt f.

Theorem 5.2 - Let P be uniformly implemented wrt $′. Let g be the base generalization of f. If P is correct wrt f, then there exists a function f′ which is the extension of f defined by (5.13), i.e.

{(c $ X,g(c $ f(X))) | X ∈ D(f) & c $ f(X) ∈ D(g)}.

Proof - Let f′ be the function computed by the loop, i.e. [P]. Since P is correct wrt f, P is correct wrt g, and f′ is a superset of both f and g. By the lemma

f′(c $ X) = f′(c $ f′(X))

for all X ∈ D(f). Since f′(X)=f(X) for X ∈ D(f) and f′(X)=g(X) for X ∈ D(g),

X ∈ D(f) & c $ f(X) ∈ D(g) -> f′(c $ X)=g(c $ f(X))

holds. Thus (5.13) is a subset of f′. Since f is a subset of f′, the union of f and (5.13) is a subset of f′. Hence this union is a function and is thus the extension of f defined by (5.13).

The following theorem is the central result presented here. The theorem formalizes the use of Lemma 5.1 in the manner suggested above, i.e. that the extension of f described in the previous theorem is a valid generalization of the original specification.

Theorem 5.3 - Let P be uniformly implemented wrt $′. Let g be the base generalization of f. If f′ is the extension of f defined by (5.13), i.e.

{(c $ X,g(c $ f(X))) | X ∈ D(f) & c $ f(X) ∈ D(g)},

then f′ is a valid generalization of f.

Proof - Suppose P is correct wrt f. We must show P is correct wrt f′. Let X ∈ D(f′). If X ∈ D(f), the loop handles the input correctly by hypothesis. If X is not in D(f), we must have X = c $ X1 where X1 ∈ D(f) and c $ f(X1) ∈ D(g). By Lemma 5.1, [P](X) = [P](c $ [P](X1)). Since P is correct wrt f this is [P](X) = [P](c $ f(X1)). By Theorem 5.1, P is correct wrt g. Using this, the equality can be written as [P](X) = g(c $ f(X1)). By the definition of f′ this implies [P](X) = f′(X). Thus P and f′ are in agreement on the input X and consequently are in agreement on any input in D(f′). Hence P is correct wrt f′ and thus f′ is a valid generalization of f.

The significance of Theorem 5.3 is that it provides a guideline for generalizing the specification of a uniformly implemented loop. If the loop is closed for the domain of the resulting specification, the generalization can then be used to prove/disprove the program correct wrt the original specification.

5.3. Applications

In this section we illustrate the use of Theorem 5.3 with a number of example programs which fall into either of two subclasses of uniformly implemented loops. The subclasses correspond to the two possible circumstances which can occur when c $ f(X) of set definition (5.13) belongs to the set D(g): the first, because ~B(c $ f(X)), and, the second, because c $ f(X) ∈ D(f). In each of these situations, the set definition (5.13) takes on a particularly simple form.

Definition 5.7 - A uniformly implemented loop satisfying
~B(X) -> ~B(c $ X)
is a Type A loop.

Observe that this condition along with (5.9) indicates that a Type A uniformly implemented loop satisfies
B(X) <-> B(c $ X),
i.e. the value of the loop predicate B is independent of a change to the data state by the operator $.

The intuition behind a Type A uniformly implemented loop is as follows. Whenever an execution of a Type A loop terminates (i.e. ~B(X) holds) and the resulting data state is modified by the operator $, the result is a new data state which, when viewed as a loop input, corresponds to zero iterations of the loop (i.e. the predicate B is still false despite the modification). This property is reflected in the following corollary.

Corollary 5.3-1 - Let P be a Type A loop. If f′ is the extension of f defined by
(5.14) {(c $ X, c $ f(X)) | X ∈ D(f)},
then f′ is a valid generalization of f.

Proof - The proof consists of showing that (5.13) and (5.14) are equivalent for a Type A loop which is correct wrt f. By Theorem 5.3, the corollary then holds. Let P be a Type A loop which is correct wrt f. A consequence of the correctness property is that ~B(f(X)) for all X ∈ D(f). Since P is a Type A loop, this implies ~B(c $ f(X)). Thus c $ f(X) ∈ D(g) and g(c $ f(X)) = c $ f(X). Consequently (5.13) and (5.14) are equivalent.

Of course, once a generalization f′ has been obtained via Corollary 5.3-1, there is no reason why that result cannot be fed back into the corollary to obtain a (possibly) further generalization f″ (using f′ for f, f″ for f′). This notion suggests

the following general case of Corollary 5.3-1.

Corollary 5.3-2 - Let P be a Type A loop. If f′ is the extension of f defined by

$$\{(c1\$(c2\$ \ldots (cn\$X) \ldots), c1\$(c2\$ \ldots (cn\$f(X)) \ldots)) \mid X \in D(f) \;\&\; n>=0\},$$

then f′ is a valid generalization of f.

Example 5.4 - Consider the following program to compute exponentiation.

```
{w=1,e>0,d>=0}
while d > 0 do
    if odd(d) then w := w * e fi;
    e := e*e; d:=d/2
    od
{w=e0 ^ d0}
```

The infix operator ^ appearing in the postcondition represents integer exponentiation. In this example, w plays the role of the key variable z, and the pair <e,d> corresponds to the variable y. We now consider wrt what operation the loop might be uniformly implemented. For any operation $′, (5.7) holds (because w does not appear in the loop predicate) as does (5.6) (because the values produced in e and d are independent of w). Furthermore, (5.5) must hold for inputs which bypass the updating of w. Thus the uniformity conditions reduce to

$$d > 0 \;\&\; odd(d) \rightarrow (c\ \$'\ w) * e = c\ \$'\ (w * e)$$

Due to its associativity, it is clear the loop is uniformly implemented wrt *, where the sets C and Z are the set of integers. Since the key variable does not appear in the loop predicate, it is necessarily a Type A loop. The specification function here is

$$f(<1,e,d>) = <e \hat{}\ d, e', d'>$$

where e>0, d>=0 and e′ and d′ are the final values computed by the loop for the variables e and d. Applying Corollary 5.3-1, the function f′ defined by

$$f'(<c*1,e,d>) = <c*(e \hat{}\ d), e', d'>$$

where e>0 and d>=0 is a valid generalization of f. Since this holds for all c, the definition of f′ can be rewritten as

$$f'(<w,e,d>) = <w*(e \hat{}\ d), e', d'>,$$

where w is an arbitrary integer, e>0 and d>=0. The generalization f′ is adequate and can thus be used to test the correctness of the program wrt the original specification. Applying (5.2), (5.3) and (5.4) from above, these necessary and sufficient verification conditions are

- the loop terminates for all e>0, d>=0,
- d=0 -> w=w*(e ^ d), and
- w*(e ^ d) is a loop constant (i.e. e0 ^ d0 = w*(e ^ d) is a loop invariant),

respectively. In Section 5.4, we will discuss a simplification of the last of these verification conditions which applies for uniformly implemented loops.

Example 5.5 [Misra 79] - The following program constructs the preorder traversal of a binary tree with root node r. The program uses a stack variable st and records the traversal in a sequence variable seq.

```
{seq=NULL, st=(r) /*stack st contains only the root node r*/}
while st ≠ EMPTY do
   p <= st; /*pop the top off the stack*/
   seq := seq || name(p); /*concatenate name of p onto seq*/
   if right(p) ≠ NIL then st <= right(p) fi; /*push onto st*/
   if left(p) ≠ NIL then st <= left(p) fi
   od
{seq=PREORDER(r)}
```

The function PREORDER(r) appearing in the postcondition is the sequence consisting of the preorder traversal of the binary tree with root node r. Let seq be the key variable. Similar reasoning to that employed in the previous example indicates here that the loop is uniformly implemented wrt ||, where the sets C and Z are the set of all sequences. It is a Type A loop. The specification function is

$$f(\langle NULL,(r)\rangle) = \langle PREORDER(r),st'\rangle.$$

Again, the ' notation is used to represent the final values of variables that are of no interest. Applying Corollary 5.3-1 we obtain

$$f'(\langle seq,(r)\rangle) = \langle seq||PREORDER(r),st'\rangle$$

as a valid generalization of f. In this case, f' is not adequate since it does not specify a behavior of the loop for arbitrary values of the stack st. We will return to this example after considering another subclass of uniformly implemented loops.

Definition 5.8 - A uniformly implemented loop satisfying

$$\tilde{B}(X) \rightarrow c \$ X \in D(f)$$

is a Type B loop.

The intuition behind a Type B uniformly implemented loop is as follows. Whenever an execution of a Type B loop terminates (i.e. $\tilde{B}(X)$ holds) and the resulting data state is modified by the operator \$, the result is a new data state which is a "valid" starting point for a new execution of the loop (i.e. this new state is in D(f)). This property is reflected in the following corollary.

Corollary 5.3-3 - Let P be a Type B loop. If f' is the extension of f defined by

$$(5.15)\quad \{(c \$ X, f(c \$ f(X))) \mid X \in D(f)\},$$

then f' is a valid generalization of f.

Proof - The proof consists of showing that (5.13) and (5.15) are equivalent for a Type B loop which is correct wrt f. By Theorem 5.3, the corollary then holds. Let P be a Type B loop which is correct wrt f. A consequence of the correctness property is that $\tilde{B}(f(X))$ for all $X \in D(f)$. Since P is a Type B loop,

this implies c $ f(X) ∈ D(f). Thus c $ f(X) ∈ D(g) and g(c $ f(X))=f(c $ f(X)). Consequently (5.13) and (5.15) are equivalent.

As before, a general case of this corollary can be stated which corresponds to an arbitrary number of its applications.

Corollary 5.3-4 - Let P be a Type B loop. If f′ is the extension of f defined by
{(c1$(c2$(...$(cn$X)...)),f(c1$f(c2$f(...$f(cn$f(X))...))))|
X ∈ D(f) & n>=0},
then f′ is a valid generalization of f.

Example 5.5 (continued) - We now consider the problem of further generalizing the derived specification in the previous example. The variable for which the loop is not closed, st, will now be the key variable. Consider an operation c $′ st that has the effect of adding an element c to the stack st. Before being more precise about this operation, we consider how the loop body works, and how its output depends on the value of the key variable st.

We observe that the loop-body behavior relies heavily on the characteristics of the node on the top of the stack. Consequently, a modification c $′ st to st which pushed a new node c onto the top of st would not cause a slight and orderly change in the result produced by the loop body and the uniformity conditions (5.5)-(5.7) would not hold. However, because the loop-body behavior seems to be independent of what lies underneath the top of the stack, we suspect the loop is uniformly implemented wrt ADDUNDER, where C is the set of binary tree nodes, Z is the set of stacks of binary tree nodes, and c ADDUNDER st is the stack that results from adding c to the bottom of st. Conditions (5.5)-(5.7) for this operation indicate that, indeed, this is the case.

Let f be the generalization f′ from the previous example. In keeping with the convention described above, since st is now the key variable, we will reverse the order in which the two variables appear in the data state, i.e. we will write <st,seq> instead of <seq,st>.

The program is a Type B uniformly implemented loop since
st=EMPTY -> <c ADDUNDER st,seq> ∈ D(f)
where c is a node of a binary tree, and specifically
(5.16) st=EMPTY -> f(<c ADDUNDER st,seq>)
=<st′,seq||PREORDER(c)>.
Applying Corollary 5.3-4, if (r,cn, ... ,c1) is an arbitrary stack (with r on top, c1 on the bottom, n>=0)
f′(<(r,cn, ... ,c1),seq>) =
f′(c1$(c2$(... $(cn$<(r),seq>) ...))) =
f(c1$f(c2$f(... $f(cn$f(<(r),seq>)) ...))) =
f(c1$f(c2$f(... $f(cn$<st′,seq||PREORDER(r)>) ...))).

Recall that st´ refers to the final value of st computed by the loop. The loop predicate indicates this will always be the value EMPTY. Hence (5.16) can be applied to this expression from inside out giving

f(c1$f(c2$f(...$<st´,seq||PREORDER(r)||PREORDER(cn)>...)))
= ... =
<st´,seq||PREORDER(r)||PREORDER(cn)|| ... ||PREORDER(c1)>.

Note that f´ now defines a loop behavior for all sequences seq and nonempty stacks st. The base generalization of f´ supplements f´ with a behavior for the empty stack st and is thus an adequate generalization.

Example 5.6 [Gries 79] - The following program computes Ackermann's function using a sequence variable s of natural numbers. The notation s(1) is the rightmost element of s and s(2) is the second rightmost, etc. The sequence s(..3) is s with s(2) and s(1) removed.

```
{s=<m,n>,m>=0,n>=0}
while size(s) ≠ 1 do
   if     s(2)=0 then s:=s(..3)||<s(1)+1>
   elseif s(1)=0 then s:=s(..3)||<s(2)-1,1>
   else               s:=s(..3)||<s(2)-1,s(2),s(1)-1> fi
   od
{s=<A(m,n)>}
```

The function A(m,n) appearing in the postcondition in Ackermann's function. The specification function is

f(<s(2),s(1)>)=<A(s(2),s(1))>.

Let s be the key variable. As the loop-body behavior is independent of the leftmost portion of s, the loop is uniformly implemented wrt |, where C is the set of natural numbers, Z is the set of nonempty sequences of natural numbers, and c|s = <c>||s. The program is also a Type B loop. By Corollary 5.3-4 (where n>1),

```
f´(<s(n),s(n-1), ... ,s(1)>) =
f´(s(n)$(s(n-1)$( ... $(s(3)$<s(2),s(1)>) ... ))) =
f(s(n)$f(s(n-1)$f( ... $f(s(3)$f(<s(2),s(1)>)) ... ))) =
f(s(n)$f(s(n-1)$f( ... $f(s(3)$<A(s(2),s(1)>) ... ))) =
f(s(n)$f(s(n-1)$f( ... $f(<s(3),A(s(2),s(1))>) ... ))) =
f(s(n)$f(s(n-1)$f( ... $<A(s(3),A(s(2),s(1)))> ... ))
        = ... =
<A(s(n),A(s(n-1), ... ,A(s(3),A(s(2),s(1))) ... )>
```

is a valid generalization of f. As in the previous example, the base generalization of this function is adequate.

5.4. Simplifying the 'Iteration Condition'

The view of WHILE loop verification presented here is one of a two step process, the first step being the discovery of an adequate valid generalization f´ of the loop specification f, the second being the proof of 3 basic conditions (i.e. (5.2)-(5.4)) based on this generalization. We have seen that the uniform nature of a loop implementation may be used in the first step as an

aid in discovering an appropriate generalization. In this section, we will exploit the same loop characteristic to substantially simplify one of the conditions which must be proven in the second step of this process.

The verification condition of interest is (5.4) above, i.e.

$$X \in D(f') \;\&\; B(X) \rightarrow f'(X)=f'(H(X)),$$

and is labeled the iteration condition in [Misra 78]. This condition assures that as the loop executes, the intermediate values of X remain in the same level set of f', i.e. the value of f' is constant across the loop iterations. Previously we argued that if P is uniformly implemented wrt $', a change in the key variable by $' causes a slight but orderly change in the result produced by H. Roughly speaking then, the behavior of H is largely independent of the key variable. If f' is chosen so as to be equally independent of the key variable, and the above condition holds for X=<z,y> where y is arbitrary but the key variable z has a specific simple value, we might expect the condition to hold for all X. Such an expectation would be based on the belief that the truth or falsity of this condition would also be largely independent of the key variable.

We formally characterize this circumstance in the following definition.

Definition 5.9 - Let P be a loop of the form described above. A generalization f' of f is represented by f iff

(5.17) $X \in D(f) \;\&\; B(X) \rightarrow f(X)=f'(H(X))$

->

(5.18) $X \in D(f') \;\&\; B(X) \rightarrow f'(X)=f'(H(X))$.

Thus if f' is represented by f, condition (5.17) can be used in place of the iteration condition (5.18) in proving the loop is correct wrt f' (and hence wrt f). The significance of this situation is that the iteration condition can be tested with the key variable constrained by initialization (as prescribed in D(f)). In practice, the result is one of having to prove a substantially simpler verification condition.

The following theorems state that the use of Corollaries 5.3-2 and 5.3-4 lead to generalizations which are represented by the original specification.

Theorem 5.4 - Let P be a Type A loop. Suppose f' is the valid generalization of f defined in Corollary 5.3-2 and suppose f' is adequate. Then f' is represented by f.

Proof - Suppose (5.17) holds and select some arbitrary X' from D(f') satisfying B(X'). Thus there exists c1, ..., cn $\in$ C, n>=0 and X $\in$ D(f) such that

$$X' = c1\$(c2\$(\ldots\$(cn\ \$\ X\ )\ldots)).$$

By the definition of a Type A loop, we must have B(X). Applying the definition of f' yields

f'(X')=c1$(c2$(...$(cn$f (X)...))
which is
=c1$(c2$(...$(cn$f'(H(X))...))
by (5.17) since B(X) holds. Since H(X) ∈ D(f') (since f' is adequate), there exists d1, ..., dm ∈ C, m>=0, and X1 ∈ D(f) such that
H(X) =d1$(d2$(...$(dm$ X1)...)).
Furthermore,
f'(H(X))=d1$(d2$(...$(dm$f(X1))...)).
Hence, continuing from above
f'(X')= c1$(...$(cn$(d1$(...$(dm$f(X1))...)))...)
which is
=f'(c1$(...$(cn$(d1$(...$(dm$ X1)...)))...))
from the definition of f'. Thus
f'(X')=f' (c1$(...$(cn$H(X))...))
which is
=f'(H(c1$(...$(cn$ X)...)))
from the uniformity condition (5.8). Hence
f'(X')=f'(H(X'))
and the theorem is proved.

Theorem 5.5 - Let P be a Type B loop. Suppose f' is the valid generalization of f defined in Corollary 5.3-4 and suppose f' is adequate. Then f' is represented by f.

Proof - Suppose (5.17) holds and select some arbitrary X' from D(f') satisfying B(X'). Thus there exists c1, ..., cn ∈ C, n>=0 and X ∈ D(f) such that
X'= c1$(c2$(...$(cn $ X)...)).
We make the assumption that B(X). Otherwise, by the definition of a Type B loop, the term cn $ X can be replaced by another X ∈ D(f). Since B(X'), this process can be continued until X' is written in the form above, with X ∈ D(f) and B(X). Applying the definition of f' yields
f'(X')=f(c1$f(c2$f(...$f(cn$f (X))...)))
which is
=f(c1$f(c2$f(...$f(cn$f'(H(X)))...)))
by (5.17) since B(X) holds. Since H(X) ∈ D(f') (since f' is adequate), there exists d1, ..., dm ∈ C, m>=0, and X1 ∈ D(f) such that
H(X) =d1$(d2$(...$(dm$X1)...)).
Furthermore,
f'(H(X))=f(d1$f(d2$f(...$f(dm$f(X1))...))).
Hence, continuing from above
f'(X')=f(c1$f(...$f(cn$f(d1$f(...$f(dm$f(X1))...)))...))
which is
=f' (c1$(...$(cn$(d1$(...$(dm$X1)...)))...))
from the definition of f'. Thus
f'(X')=f' (c1$(...$(cn$H(X))...))
which is
=f'(H(c1$(...$(cn$ X)...)))
from the uniformity condition (5.8). Hence
f'(X')=f'(H(X'))

and the theorem is proved.

Example 5.7 - Consider the exponentiation program of Example 5.4. The generalization obtained from Corollary 5.3-2 is

$$f'(\langle w,e,d\rangle)=\langle w*(e\hat{}d),e',d'\rangle,$$

where $e>0$, $d\geq 0$. Since f' is represented by f, the iteration condition corresponding to (5.17)

$$e>0 \;\&\; d>0 \;\&\; \text{odd}(d) \rightarrow e\hat{}d = e*((e*e)\hat{}(d/2)) \;\&$$
$$e>0 \;\&\; d>0 \;\&\; \tilde{}\text{odd}(d) \rightarrow e\hat{}d = 1*((e*e)\hat{}(d/2))$$

can be used in place of that corresponding to (5.18)

$$e>0 \;\&\; d>0 \;\&\; \text{odd}(d) \rightarrow w*(e\hat{}d) = (w*e)*((e*e)\hat{}(d/2)) \;\&$$
$$e>0 \;\&\; d>0 \;\&\; \tilde{}\text{odd}(d) \rightarrow w*(e\hat{}d) = w*((e*e)\hat{}(d/2)).$$

The benefits of this simplification are more striking for more complex types of key variables. To illustrate, consider the program to compute Ackermann's function in Example 5.6. Applying Corollary 5.3-4 to the base generalization g of f yields the generalization defined by

$$n>1 \rightarrow f'(\langle s(n),s(n-1),\ldots,s(1)\rangle) = \langle A(s(n),A(s(n-1),\ldots,A(s(3),A(s(2),s(1)))\ldots))\rangle$$

and

$$f'(\langle s(1)\rangle)=\langle s(1)\rangle.$$

Since f' is represented by g, the iteration condition

$$s(2)=0 \rightarrow \langle A(s(2),s(1))\rangle=\langle s(1)+1\rangle \;\&$$
$$s(2)\neq 0 \;\&\; s(1)=0 \rightarrow \langle A(s(2),s(1))\rangle=\langle A(s(2)-1,1)\rangle \;\&$$
$$s(2)\neq 0 \;\&\; s(1)\neq 0 \rightarrow \langle A(s(2),s(1))\rangle=\langle A(s(2)-1,A(s(2),s(1)-1))\rangle$$

can be used in place of

$$n>1 \;\&\; s(2)=0 \rightarrow \langle A(s(n),A(s(n-1),\ldots,A(s(3),A(s(2),s(1)))\ldots))\rangle = \langle A(s(n),A(s(n-1),\ldots,A(s(3),s(1)+1)\ldots))\rangle \;\&$$
$$n>1 \;\&\; s(2)\neq 0 \;\&\; s(1)=0 \rightarrow \langle A(s(n),A(s(n-1),\ldots,A(s(3),A(s(2),s(1)))\ldots))\rangle = \langle A(s(n),A(s(n-1),\ldots,A(s(3),A(s(2)-1,1))\ldots))\rangle \;\&$$
$$n>1 \;\&\; s(2)\neq 0 \;\&\; s(1)\neq 0 \rightarrow \langle A(s(n),A(s(n-1),\ldots,A(s(3),A(s(2),s(1)))\ldots))\rangle = \langle A(s(n),A(s(n-1),\ldots,A(s(3),A(s(2)-1,A(s(2),s(1)-1)))\ldots))\rangle.$$

5.5. Recognizing Uniformly Implemented Loops

Although the problem of recognizing uniformly implemented loops is in general an unsolvable problem, the following guidelines seem useful in a large number of situations.

Recognizing uniformly implemented loops can be viewed as a search for an operation wrt which the loop is uniformly implemented. In practice, condition (5.5) is the most demanding constraint on this operation. An effective strategy, therefore, is to use (5.5) as a guideline to suggest candidate operations. Conditions (5.6) and (5.7) must be proven to show the loop is uniformly implemented wrt some particular candidate.

Often the modification to the key variable z in the loop body is performed by a statement of the form

$$z := z \# e(y)$$

for some dyadic operation # and function e. In this case, condition (5.5) suggests the loop may be uniformly implemented wrt # or some directly related operation. For example, if # is associative, condition (5.5) holds for #. If # satisfies

$$(a \# b) \# c = (a \# c) \# b$$

(e.g. subtraction), and an inverse #' of # exists satisfying

$$a \# b = c \iff b \#' c = a$$

(e.g. addition if # is subtraction), condition (5.5) holds for #'.

Another commonly occurring case is when the future values of the key variable z are independent of y, i.e.

$$h'(z,y1) = h'(z,y2)$$

for all z, y1 and y2. This situation arises most frequently when z is some data structure which varies dynamically as the loop iterates. Typically, there exists some particular aspect or portion of the data structure (e.g. the top of a stack, the end of a sequence, the leaf nodes in a tree) which guides its modification. A useful heuristic which can be employed in this circumstance is to consider only operations which maintain (i.e. keep invariant) this particular aspect of the data structure. Selecting such an operation $' guarantees that the "change" experienced by the data structure in the loop body will be independent of any modification $' and thus insures condition (5.5) holds.

In any case, recognizing uniformly implemented loops and determining the operation wrt which they are uniformly implemented is often facilitated if the intended effect of the loop body (as regards the key variable) is documented in the program source text. Such documentation abstracts what the loop body does from the method employed to achieve this result and thus makes analysis of the loop as a whole easier.

To illustrate, consider the following program to compute the maximum value in a subarray a[i..n] of natural numbers:

```
{m=0}
while i <= n do
   if m < a[i] then m := a[i] fi;
   i := i + 1
   od
{m=MAXIMUM(a,i0,n)}.
```

If the effect on m in the loop body were documented as

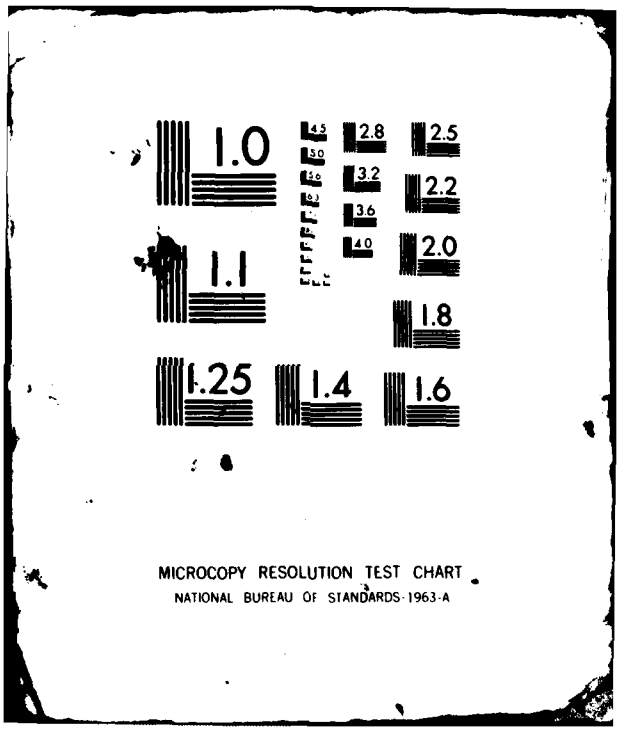
$$m := MAX(m, a[i]),$$

its updating would be of the form m := m # a[i] and the heuristic discussed above could be employed to help determine that the loop is uniformly implemented wrt # = MAX.

AD-A113 040 MARYLAND UNIV COLLEGE PARK COMPUTER SCIENCE CENTER F/6 9/2
AN INVESTIGATION OF FUNCTIONAL CORRECTNESS ISSUES.(U)
1982 D D DUNLOP F49620-80-C-0001
UNCLASSIFIED CSC-TR-1135 AFOSR-TR-82-0263 NL

2 2

END
DATE FILMED
4 82
DTIC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

5.6. Related Work

The first work on generalizing functional specifications for loops appears in [Basu & Misra 76]. These results are refined in [Misra 78] and are studied in considerable detail in [Misra 79]. The major contribution of this research seems to be the identification of two loop classes or schemas which are "naturally provable." The first class is called the accumulating loop schema and can be viewed as a (commonly occurring) special case of the Type A loops discussed above. Specifically, a program in the accumulating loop schema with associative binary operation $' in the sense of [Basu & Misra 76] is necessarily uniformly implemented wrt $' and meets the criterion for a Type A loop presented above.

The second of these classes is called the structured data schema. A loop in this class is uniformly implemented wrt an operator which adds an element to the data structure being processed in such a way that it is not the "next" element to be removed from the structure (e.g. recall the use of ADDUNDER in the tree traversal example). A loop in this class necessarily meets the criterion for a Type B loop presented above. The program to compute Ackermann's function does not fit in the structured data schema. We remark that the analysis presented in this chapter relies on the loop body computing a function, i.e. it relies on the loop body being deterministic. Consequently, the above comments do not apply to the non-deterministic structured data loops analyzed in [Misra 79].

In [Misra 79] the author states that the important common feature between these program classes is that " ... they act upon data in a 'uniform' manner; changes in the input data lead to certain predictable changes in the result obtained." The work described in this chapter can be viewed as an attempt to characterize this commonality and to generalize the work in [Misra 79] based on this characterization.

More recently, [Basu 80] considers the problem of generalizing loop specifications and uses the idea of a loop being "uniform over a linear data domain." One difference between this work and our notion of a uniformly implemented loop is that Basu considers only programs in the accumulating loop schema (in the sense of [Basu & Misra 76] without the closure requirement). More importantly, Basu's idea of uniform behavior is based on the behavior of the loop as a whole and seems to be largely independent of the loop body. Our approach relies solely on the characteristics of the loop body.

Misra points out in [Misra 78, Misra 79] that the iteration condition for his structured data schema can be simplified in a manner similar to that presented here; our results show that the same simplification can be applied to his accumulating loop schema. Again, an appropriate view of the research described in this

chapter is one of generalizing this earlier work by investigating the theory which underlies these phenomenon.

5.7. Discussion

It is felt that a critical aspect of reading, understanding and verifying program loops is generalizing the behavior of a loop over a restricted set of inputs to that over a more general set of inputs. The view of this generalization process presented in this chapter is one of ascertaining how changes in values of particular input variables affect the subsequent computation of the loop. This process is facilitated if these changes correspond to particularly simple modifications in the result produced by the loop body.

Of course, the simplest possible modifications in the result produced by the loop body would be no modifications at all, i.e. the output of the loop body (and hence the loop) is completely independent of changes in these input variables. This situation, however, occurs rarely in practice since it implies that the input values of these variables serve no purpose in view of the intended effect of the loop. It is felt that the definition of a uniformly implemented loop presented above is the "next best" alternative, and yet a large number of commonly occurring loops seem to possess this property. The definition states that in terms of the execution of the loop body, prescribed changes in the input value of the key variable affect only the final value of the key variable; all other final values are independent of the change. Just as importantly, the modification caused in the final value of the key variable is necessarily the same as the change in its corresponding input value. This property is analogous to that possessed by a function of 1 variable with unit slope in analytic geometry: increasing the input argument by some constant causes the function value to be increased by exactly the same quantity. Taken together, these factors account for the pleasing symmetry between $ and H in condition (5.8).

Viewed as a verification technique for uniformly implemented loops, the procedure described above can be thought of as transforming the problem of discovering the general loop specification into the problem of discovering the operation with respect to which the loop is uniformly implemented. Clearly, this is of no benefit if the latter is no easier to solve than the former. In many cases, however, it seems that simple syntactic checks are sufficient for identifying this operation. For example, in the tree traversal program, the fact that the loop body does not test the stack for emptiness [Basu & Misra 76] is a sufficient condition for the loop being uniformly implemented with respect to ADDUNDER.

It is felt that the notion of uniformly implemented loops may have an application in the program development process. Specifically, when designing an initialized loop to compute some

function, the programmer should attempt to construct the loop in such a way that it is uniformly implemented with respect to some easily stated operation. Our work indicates that these loops are susceptible to a rather routine form of analysis. Furthermore, implementing a loop in a uniform fashion requires maintaining a certain amount of independence between program variables (or perhaps portions of program variables in the case of structures) and a simple dependence between the input/output values computed by the loop body. Such programs are desirable since the ease with which a loop can be understood depends largely on the complexity of the interactions and interconnections among program variables. We remark that the question of whether a given program is "well structured" has been viewed largely as a <u>syntactic</u> issue (e.g. use of a restricted set of control structures); we offer the definition of a uniformly implemented loop as an attempt at a characterization of a <u>semantically</u> well structured program.

6. Summary and Concluding Remarks

The purpose of this research has been an investigation of a number of issues involved with the application of the functional correctness methodology. In Chapter 2, we defined a functional correctness technique and addressed the issue of the strategy employed in decomposing composite programs in the proof process.

Chapters 3-5 dealt largely with the issue of verifying programs containing loops with undocumented or inadequate intended functions. In Chapter 3, the idea of a reduction hypothesis was introduced. The use of a reduction hypothesis in a proof of correctness eliminates the need for the intended loop function. As an alternative approach, a heuristic procedure for constructing intended loop functions was described in Chapter 4. Finally, in Chapter 5, a class of loops was described for which inadequate intended loop functions can be extended or generalized in a systematic manner.

The most promising directions for future research seem to stem from the results presented in Chapter 5. While each uniformly implemented loop we have studied does exhibit a "reasonable" form of behavior, there do exist "reasonable" initialized loop programs which do not satisfy our definition. This situation invariably occurs when initialized data is used in an indirect manner (e.g. as an array index or pointer). A characterization of uniform loop behavior with respect to indirect data would serve as a useful complement to the results described in Chapter 5.

A second direction for future research is the development of a program design/implementation methodology based on the uniformly implemented loop. At the point of designing an initialized loop program, the methodology would suggest the programmer first formulate an appropriate operation, and then proceed to detail the loop in a way which insured that it was uniformly implemented with respect to that operation. We argued previously that uniformly implemented loops are "well structured" programs; such a methodology might be a valuable contribution to what has come to be known as "structured programming."

7. References

[Basili & Noonan 80]
Basili, V. R. and Noonan, R. E. A Comparison of the Axiomatic and Functional Models of Structured Programming, IEEE Transactions on Software Engineering, Vol. SE-6, Sept. 1980, pp. 454-464.

[Basu 80]
Basu, S. A Note on Synthesis of Inductive Assertions, IEEE Transactions on Software Engineering, Vol. SE-6, Jan. 1980, pp. 32-39.

[Basu & Misra 75]
Basu, S. and Misra, J. Proving Loop Programs, IEEE Transactions on Software Engineering, Vol. SE-1, March 1975, pp. 76-86.

[Basu & Misra 76]
Basu, S. K. and Misra, J. Some Classes of Naturally Provable Programs, Proc. 2nd International Conf. on Software Engg., San Francisco, Oct. 1976, pp. 400-406.

[Dijkstra 76]
Dijkstra, E. W. A Discipline of Programming, Prentice-Hall, 1976.

[Ellozy 81]
Ellozy, H. The Determination of Loop Invariants for Programs with Arrays, IEEE Transactions on Software Engineering, Vol. SE-7, March 1981, pp. 197-206.

[Floyd 67]
Floyd, R. W. Assigning Meanings to Programs, Proceedings of a Symposia in Applied Mathematics, 19, 1967, pp. 19-32.

[Gries 79]
Gries, D. Is Sometime Ever Better Than Alway?, Transactions on Programming Languages and Systems, Vol. 1, Oct. 1979, pp. 258-265.

[Hoare 69]
Hoare, C. A. R. An Axiomatic Basis for Computer Programming, CACM, Vol. 12, Oct. 1969, pp. 576-583.

[Katz & Manna 73]
Katz, S. and Manna, Z. A Heuristic Approach to Program Verification, Proc. 3rd Int. Joint Conf. Artificial Intell., Stanford, CA 1973, pp. 500-512.

[Katz & Manna 76]
Katz, S. and Manna, Z. Logical Analysis of Programs, CACM, Vol. 19, April 1976, pp. 188-206.

[King 80]
King, J. Program Correctness: On Inductive Assertion Methods, IEEE Transactions on Software Engineering, Vol. SE-6, Sept. 1980, pp. 465-479.

[Linger, Mills & Witt 79]
Linger, R. C., Mills, H. and Witt, B. I. Structured Programming Theory and Practice, Addison-Wesley, 1979.

[McCarthy 62]
McCarthy, J. Towards a Mathematical Science of Computation. In: Popplewell, C.M. (ed.): Proc. IFIP Congress 62, North-Holland, Amsterdam, 1963, pp. 21-28.

[McCarthy 63]
McCarthy, J. A Basis for a Mathematical Theory of Computation. In: Brafford, P., and Hirschberg, D. (eds.): Computer Programming and Formal Systems, Amsterdam, North Holland, 1963, pp. 33-70.

[Manna 71]
Manna, Z. Mathematical Theory of Partial Correctness, J. Computer System Sci., Vol. 5, June 1971, pp. 239-253.

[Manna & Pnueli 70]
Manna, Z. and Pnueli, A. Formalization of Properties of Functional Programs, JACM, Vol. 17, July 1970, pp. 555-569.

[Manna & Waldinger 70]
Manna, Z. and Waldinger, R. Towards Automatic Program Synthesis, Stanford Artificial Intelligence Project, Memo AIM-127, July 1970.

[Mills 72]
Mills, H. D. Mathematical Foundations for Structured Programming, IBM Federal Systems Division, FSC 72-6012, 1972.

[Mills 75]
Mills, H. D. The New Math of Computer Programming, CACM, Vol. 18, Jan. 1975, pp. 43-48.

[Misra 77]
Misra, J. Prospects and Limitations of Automatic Assertion Generation for Loop Programs, SIAM J. Comput., Vol. 6, Dec. 1977, pp. 718-729.

[Misra 78]
Misra, J. Some Aspects of the Verification of Loop Computations, IEEE Transactions on Software Engineering, Vol. SE-4, Nov. 1978, pp. 478-486.

[Misra 79]
Misra, J. Systematic Verification of Simple Loops, Univer-

sity of Texas Technical Report TR-97, March 1979.

[Morris & Wegbreit 77]
Morris, J. H. and Wegbreit, B. Subgoal Induction, CACM, Vol. 20, April 1977, pp. 209-222.

[Strachey 64]
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.

[Wegbreit 74]
Wegbreit, B. The Synthesis of Loop Predicates, CACM, Vol. 17, Feb. 1974, pp. 102-112.

[Wegbreit 77]
Wegbreit, B. Complexity of Synthesizing Inductive Assertions, JACM, Vol. 24, July 1977, pp. 504-512.