

A Heuristic For
Deriving Loop Functions

Douglas D. Dunlop and Victor R. Basili

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

D. Dunlop is with Intermetrics, Inc., 4733 Bethesda Avenue, Suite 415, Bethesda, MD 20814.

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.

ABSTRACT

The problem of analyzing an initialized loop and verifying that the program computes some particular function of its inputs is addressed. A heuristic technique for solving these problems is proposed that appears to work well in many commonly occurring cases. The use of the technique is illustrated with a number of applications. An attribute of initialized loops is identified that corresponds to the "effort" required to apply this method in a deterministic (i.e. guaranteed to succeed) manner. It is explained that in any case, the success of the proposed heuristic relies on the loop exhibiting a "reasonable" form of behavior.

KEYWORDS and PHRASES: program verification, initialized loop programs, loop functions, constraints

CR CATEGORIES: 5.24

1. Introduction

In this report, 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 paper, 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 method 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 that is taken in this paper is to invent an hypothesis concerning the function of the WHILE loop, i.e., its general input/output behavior. Once this has been done, the loop can be proven/disproven correct with respect to the hypothesis using standard techniques [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 method is similar to this technique in that we attempt to infer the general behavior of the loop from several sample loop behaviors. In contrast, 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.

2. Constraint Functions and Loop Functions

The verification problem is represented as follows:

```

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

```

X represents the state of the program and may be viewed as a vector containing values for each of the variables in the program. K and H are state-to-state functions corresponding to the effects of the initialization and loop body respectively. B is a predicate over the state. The program is to produce in the output variable(s) V, a function f of the input state X0. D(f) stands for the domain of the function f, i.e. the set of states for which a value of f is defined.

If S is the set of all conceivable program states and T is the set of values that the variables V may assume, f has the functionality $f : S \rightarrow T$. In order to verify a program of this form, we choose to find a function $g : S \rightarrow T$ that 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 states generated by each loop iteration. If this is the case, the loop is said to be closed [Basu & Misra 75, Misra 78] 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 initialized loop 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 two approaches attack the problem of synthesizing g from different directions. The "top down" approach works from the function specification, the "bottom up" approach works from the code. 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 (i.e., specification and code vs. just code) when synthesizing the hypothesis. The disadvantage of such an approach is that it may not be as well-suited to disproving programs. This is because to disprove a program, the verifier must employ an argument that shows that there does not exist a valid hypothesis. The method described in this paper is based on the "top down" approach. We will return to a discussion of this advantage and disadvantage later.

Assuming the above program is correct, several properties of g can be deduced. First

$$(1) \quad X0 \in D(f) \rightarrow f(X0)=g(K(X0)).$$

That is, for inputs satisfying the program precondition, the initialization followed by the loop yields the desired result. Secondly, since the loop computes g , the "iteration condition" [Misra 78]

$$B(X1) \rightarrow g(X1)=g(H(X1))$$

of the standard technique for showing the loop computes g must hold [Mills 75, Morris & Wegbreit 77, Misra 78]. This implies

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

Combining with (1) yields

$$(2) \quad X0 \in D(f) \ \& \ B(K(X0)) \rightarrow f(X0)=g(H(K(X0))).$$

Predicates (1) and (2) can be rewritten using a new, universally quantified variable as follows:

(1') $X0 \in D(f) \ \& \ X=K(X0) \ \rightarrow \ g(X)=f(X0)$

and

(2') $X0 \in D(f) \ \& \ B(K(X0)) \ \& \ X=H(K(X0)) \ \rightarrow \ g(X)=f(X0).$

If the above program is correct, then the function g that the loop computes must satisfy (1') and (2'). We call (1') and (2') constraints, since they serve as constraints or requirements on the loop function g . (1') views the program as initialization followed by the loop -- as a single entity. (2') views the program as initialization followed by the loop body followed by the loop -- again as a single entity. In a sense, we are using the well known fact that if condition $B(X)$ is true, then the programs

<u>while</u> $B(X)$ <u>do</u>	$X := H(X);$
$X := H(X)$	<u>while</u> $B(X)$ <u>do</u>
<u>od</u> ;	$X := H(X)$
	<u>od</u> ;

are equivalent for the input X . One could derive other constraints as well (and we will, later on), by looking at the loop as two separate iterations followed by the loop, etc.

Thus far, we have developed two constraints (1') and (2') that the loop function g must satisfy. The constraints represent subsets of g . We are interested in them because, if the first few iterations of the loop are in some sense representative of all iterations, then it might be possible to generalize the constraints to g in a relatively straightforward manner. We believe that this is indeed the case, and we have developed some promising heuristics for it. We present these heuristics in the next section, but first let us look at a specific example to make these ideas more concrete.

Example 1 - The following program performs $z := v0*k$ for $v0 \geq 0$:

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

For this program, the two constraints (1') and (2') are

C1: $v=v0 \ \& \ v0 \geq 0 \ \& \ z=0 \ \rightarrow \ g(z,v,k)=v0*k$
C2: $v0 > 0 \ \& \ v=v0-1 \ \& \ z=k \ \rightarrow \ g(z,v,k)=v0*k.$

Here, we have used the specific variables to represent the state. The domain $D(f)$ of the program is given by the precondition. The predicate $X=K(X0)$ of constraint (1') is $z=0$. The function f computed by the loop is $v0*k$. Note that the function g is written as a function

of all the variables used in the loop

Now let us try to generalize constraints C1 and C2 in order to arrive at a possible loop function g . First, since v_0 does not occur as an argument of g , it may help to eliminate it. This is possible because the antecedent of each constraint completely defines v_0 for that constraint. We therefore rewrite C1 and C2 as

$$\begin{aligned} \text{C1: } & v \geq 0 \ \& \ z=0 \ \rightarrow \ g(z,v,k)=v*k \\ \text{C2: } & v \geq 0 \ \& \ z=k \ \rightarrow \ g(z,v,k)=(v+1)*k. \end{aligned}$$

Next, note that argument z of g does not appear to help define g -- it does not appear in the function expressions $v*k$ and $(v+1)*k$. The appearance of $z=k$ in the antecedent of C2 indicates that we might be able to introduce z into the function expressions. First, rewrite $(v+1)*k$ as $v*k+k$:

$$\text{C2: } \ v \geq 0 \ \& \ z=k \ \rightarrow \ g(z,v,k)=v*k+k.$$

Then, an occurrence of k can be replaced by z , yielding one of the two possibilities:

$$v \geq 0 \ \rightarrow \ g(z,v,k)=v*z+k \quad \text{and} \quad v \geq 0 \ \rightarrow \ g(z,v,k)=v*k+z.$$

Both are generalizations of C2, but only the second is also a generalization of C1, so we take this latter function as the definition of the function g . With it, we can prove the above program using the standard techniques of [Mills 75, Misra 78].

This example shows how the constraints derived from a loop with initialization can be generalized to the loop function g . At first glance, the general problem of generalizing a set of constraints to some desired function may seem too difficult. However, in the limited context where the constraints are derived from an initialized loop and the function to be derived is the loop function, it may be possible to develop heuristics for generalizing a function g from a set of constraints. The methods of computation of many loops that occur in practice have enough similarities to make such heuristics useful.

In the next section we introduce a four-step method for deriving the function g from the constraints. This method was actually just used on the previous example, and we will illustrate its use on many other examples in this paper.

3. The Technique

We describe the four steps using the multiplication program of Example 1:

```

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

```

Step 1 : RECORD - This step consists of recording the constraints using the templates (1') and (2'). As a notational convenience, we dispense with the state notation and use program variables (possibly subscripted by 0 to denote their initial values) in these constraints. The terms $X0 \in D(f)$ and $f(X0)$ come from the pre and postcondition 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. The constraints for the program are as follows:

C1: $v0 \geq 0 \ \& \ v=v0 \ \& \ z=0 \ \rightarrow \ g(z,v,k)=v0*k$
C2: $v0 > 0 \ \& \ v=v0-1 \ \& \ z=k \ \rightarrow \ g(z,v,k)=v0*k.$

We make the following comments. First, g is defined as a function of each program variable that occurs in the loop predicate or loop body. Second, note that in C2, the term $v0 > 0$ captures both $X0 \in D(f)$ (i.e. $v0 \geq 0$) and $B(K(X0))$ (i.e. $v0 \neq 0$). As a final remark, we will use the phrase function expression to refer to the term in the consequent of a constraint that defines the value of g (e.g. $v0*k$ in both C1 and C2 above).

Step 2 : SIMPLIFY - All variables that appear in the constraint but not in the argument list for g must eventually be eliminated from the constraint. On occasion, it is possible to solve for the value of such a variable in the antecedent and substitute the equivalent expression for it throughout the constraint. To illustrate, in 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 to

C1: $v \geq 0 \ \& \ z=0 \ \rightarrow \ g(z,v,k)=v*k.$

In a similar manner, the second constraint can be SIMPLIFIED to (using $v0=v+1$)

C2: $v \geq 0 \ \& \ z=k \ \rightarrow \ g(z,v,k)=(v+1)*k.$

Step 3 : REWRITE - Variables that appear in the argument list for g but not in the function expression are candidates to be introduced into the function expression. Each of these variables will be bound to a term in the antecedent of the constraint. 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 include these terms

into the function expression. In the following step (see below), the result of REWRITE will then be used to introduce the necessary variables into the function expression. To illustrate, consider the above SIMPLIFIED C2. 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 antecedent. 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

$$C2: v \geq 0 \ \& \ z=k \ \rightarrow \ g(z,v,k)=v*k+k.$$

Step 4 : SUBSTITUTE - In steps 2 and 3, the constraints are massaged into equivalent constraints 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 that is known to hold when m has the value 17. Viewed in this light, the purpose of the constraint C2 is to obtain a relationship that 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 variable z. The two possible substitutions lead to the following generalizations:

$$v \geq 0 \ \rightarrow \ g(z,v,k)=v*k+z$$

and

$$v \geq 0 \ \rightarrow \ g(z,v,k)=v*z+k.$$

Both of these (necessarily) imply (i.e., are generalizations 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

$$v \geq 0 \ \rightarrow \ g(z,v,k)=v*k+z$$

for the behavior of the loop

```

while v ≠ 0 do
  z := z + k;
  v := v - 1;
od.

```

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 $v \geq 0$,
- $v=0 \rightarrow z=v*k + z$, and
- $v*k + z$ is a loop constant (i.e. $v_0*k_0 + z_0 = v*k + z$ 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 constraints 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: $v \geq 0 \ \& \ z=k \rightarrow g(z,v,k) =$
 $v*k + 3*k + k*k*(v-7)/(4*k) + k*k*k/(k*k)$
 $- k*k*k*(v-7)/(4*k*k) - k*k*k*3/(k*k)$

and

$v \geq 0 \rightarrow g(z,v,k) =$
 $v*k + 3*z + z*z*(v-7)/(4*k) + z*z*z/(k*k)$
 $- z*z*z*(v-7)/(4*k*k) - z*z*z*3/(k*k)$

respectively, where "/" denotes an integer division (with truncation) infix operator that 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 antecedent of C2 that 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 applications of this technique.

4. Applications

Example 2 - The following program computes powers of integers. This example serves to illustrate the use of the technique when the loop body contains several paths:

```
{c=c0 & d=d0 & d0>=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 first constraint is easily RECORDED:

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

and SIMPLIFIES to

$$C1: \ d \geq 0 \ \& \ w=1 \ \rightarrow \ g(w,c,d)=c^{**}d.$$

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

$$d0 > 0 \ \& \ \text{odd}(d0) \ \& \ w=c0 \ \& \ c=c0*c0 \ \& \ d=d0/2 \ \rightarrow \ g(w,c,d)=c0^{**}d0$$

which SIMPLIFIES to

$$C2a: \ d \geq 0 \ \& \ c=w*w \ \rightarrow \ g(w,c,d)=w^{**}(d*2+1).$$

The constraint corresponding to the other loop-body path is

$$d0 > 0 \ \& \ \sim\text{odd}(d0) \ \& \ w=1 \ \& \ c=c0*c0 \ \& \ d=d0/2 \ \rightarrow \ g(w,c,d)=c0^{**}d0$$

and SIMPLIFIES to

$$d \geq 0 \ \& \ w=1 \ \& \ \text{SQUARE}(c) \ \rightarrow \ g(w,c,d)=\text{SQRT}(c)^{**}(d*2)$$

i.e.

$$C2b: \ d \geq 0 \ \& \ w=1 \ \& \ \text{SQUARE}(c) \ \rightarrow \ g(w,c,d)=c^{**}d$$

where $\text{SQUARE}(x)$ = 'x is a perfect square', and $\text{SQRT}(x)$ = 'the square root of the perfect square x'. The term $\text{SQUARE}(c)$ is necessary in the antecedent since c is necessarily a perfect square in the antecedent of the unSIMPLIFIED constraint. Note that C2b is implied by C1 and hence is of no additional help in characterizing the general loop function. The heuristic suggested in REWRITE is to rewrite the constraint 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

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

This function is in agreement with 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 that bypasses the updating of the initialized data (C2b) is implied by 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 constraints 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 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 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(x) is TRUE iff x 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;
  /* s := s U SONS(e) */
  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 is

$$C1: \sim\text{empty}(t) \ \& \ n=0 \ \& \ s=\{t\} \ \rightarrow \ g(n,s)=\text{NODES}(t).$$

Rather than considering each of the four possible paths through the loop body individually, we use the abstraction for the combined effect of the two IF statements

$$s := s \cup \text{SONS}(e),$$

where $\text{SONS}(x)$ is the set of 0, 1 or 2 nonempty subtrees of x . Applying this, the second constraint is

$$C2: \sim\text{empty}(t) \ \& \ n=1 \ \& \ s=\text{SONS}(t) \ \rightarrow \ g(n,s)=\text{NODES}(t).$$

In order to introduce n and s into the function expression for $C2$, we choose to REWRITE this expression using the recursive definition that $\text{NODES}(x)$ 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

$$1+\text{SUM}(x,\text{SONS}(t),\text{NODES}(x))$$

where $\text{SUM}(A,B,C)$ stands for the summation of C over all $A \in B$. Applying SUBSTITUTE in the obvious way yields

$$\sim\text{empty}(t) \ \rightarrow \ g(n,s)=n+\text{SUM}(x,s,\text{NODES}(x))$$

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 $\sim\text{empty}(t)$ that characterizes the domain 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 $\text{empty}(t)$? The answer is that it obviously cannot; the above function is simply equivalent to

$$g(n,s)=n+\text{SUM}(x,s,\text{NODES}(x)).$$

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

As a second point, in Example 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

$$g(n,s)=n+\text{SUM}(x,s,\text{NODES}(x))$$

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

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)). \end{aligned}$$

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 \langle and \rangle to construct sequences, i.e. a sequence s consisting of n elements will be written $\langle s(n), \dots, s(2), s(1) \rangle$.

```

{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 is

$$C1: m \geq 0 \ \& \ n \geq 0 \ \& \ s = \langle m, n \rangle \rightarrow g(s) = \langle A(m, n) \rangle.$$

The second constraints corresponding to the 3 paths through the loop body are

$$\begin{aligned} C2a: \quad m=0 \ \& \ n \geq 0 \ \& \ s = \langle n+1 \rangle & \rightarrow g(s) = \langle A(m, n) \rangle \\ C2b: \quad m > 0 \ \& \ n = 0 \ \& \ s = \langle m-1, 1 \rangle & \rightarrow g(s) = \langle A(m, n) \rangle \\ C2c: \quad m > 0 \ \& \ n > 0 \ \& \ s = \langle m-1, m, n-1 \rangle & \rightarrow g(s) = \langle A(m, n) \rangle. \end{aligned}$$

REWRITING these 3 based on the above definition of A yields

$m=0 \ \& \ n \geq 0 \ \& \ s = \langle n+1 \rangle \quad \rightarrow \ g(s) = \langle n+1 \rangle$
 $m > 0 \ \& \ n = 0 \ \& \ s = \langle m-1, 1 \rangle \quad \rightarrow \ g(s) = \langle A(m-1, 1) \rangle$
 $m > 0 \ \& \ n > 0 \ \& \ s = \langle m-1, m, n-1 \rangle \rightarrow \ g(s) = \langle A(m-1, A(m, n-1)) \rangle.$

SUBSTITUTING here yields

$s = \langle s(1) \rangle \quad \rightarrow \ g(s) = \langle s(1) \rangle$
 $s = \langle s(2), s(1) \rangle \quad \rightarrow \ g(s) = \langle A(s(2), s(1)) \rangle$
 $s = \langle s(3), s(2), s(1) \rangle \quad \rightarrow \ g(s) = \langle A(s(3), A(s(2), s(1))) \rangle.$

Note that the second of these implies C1. The 3 together seem to suggest the general loop behavior (where $n > 1$)

$$g(\langle s(n), s(n-1), \dots, s(1) \rangle) = \langle A(s(n), A(s(n-1), \dots, A(s(2), s(1)) \dots)) \rangle.$$

We remark that in the first 3 examples, the heuristic resulted in a loop function that was sufficiently general (i.e. the loop was closed for the domain of the inferred function). Example 4 illustrates that this does not always occur. The loop function heuristic is helpful in the example in that SUBSTITUTE 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 5 - Let v be a one dimensional array of length $n > 0$ that 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=AMAX(v)}

```

The notation $AMAX(v)$ appearing in the postcondition stands for the largest element of the array v . The following constraints are obtained

C1: $m=0 \ \& \ i=1 \rightarrow g(m, i, v, n) = AMAX(v)$
C2: $m=v[1] \ \& \ i=2 \rightarrow g(m, i, v, n) = AMAX(v).$

Noticing the appearance of $v[1]$ and 2 in C2, we REWRITE $AMAX(v)$ in C2 as $MAX(v[1], AMAX(v[2:n]))$, 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 obtained by applying SUBSTITUTE,

$$g(m, i, v, n) = MAX(m, AMAX(v[i:n])),$$

agrees with C1.

Example 6 - If p is a pointer to a node in a binary tree, let $IN(p)$ be the sequence of pointers that point to the nodes in an inorder traversal of the binary tree pointed to by p . The following program constructs $IN(p)$ 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 , $IN(p)$ 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 = IN(rt)}.

```

Up until now, we have attempted to infer a general loop function from two constraints. 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 constraints and it proves helpful to obtain a third constraint. Functions $C1$ and $C2$ correspond to 0 and 1 loop-body executions, respectively. The third constraint $C3$ will correspond to 2 loop-body executions. We will use the notation (e_1, \dots, e_n) for a stack containing the elements e_1, \dots, e_n from top to bottom. The constraints for this program are

```

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

```

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

C2': $rt \neq \text{NIL} \ \& \ p = l(rt) \ \& \ stk = (rt) \ \& \ vs = \langle \rangle \ \rightarrow$
 $g(p, stk, vs) = \text{IN}(l(rt)) \ || \ \langle rt \rangle \ || \ \text{IN}(r(rt))$
 C3a': $rt \neq \text{NIL} \ \& \ l(rt) \neq \text{NIL} \ \& \ p = l(l(rt)) \ \& \ stk = (l(rt), rt) \ \& \ vs = \langle \rangle \ \rightarrow$
 $g(p, stk, vs) = \text{IN}(l(l(rt))) \ || \ \langle l(rt) \rangle \ || \ \text{IN}(r(l(rt)))$
 $|| \ \langle rt \rangle \ || \ \text{IN}(r(rt))$
 C3b': $rt \neq \text{NIL} \ \& \ l(rt) = \text{NIL} \ \& \ p = r(rt) \ \& \ stk = \text{EMPTY} \ \& \ vs = \langle rt \rangle \ \rightarrow$
 $g(p, stk, vs) = \langle rt \rangle \ || \ \text{IN}(r(rt)).$

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

$stk = (e1) \ \& \ vs = \langle \rangle \ \rightarrow g(p, stk, vs) = \text{IN}(p) \ || \ \langle e1 \rangle \ || \ \text{IN}(r(e1))$
 $stk = (e1, e2) \ \& \ vs = \langle \rangle \ \rightarrow g(p, stk, vs) = \text{IN}(p) \ || \ \langle e1 \rangle \ || \ \text{IN}(r(e1))$
 $|| \ \langle e2 \rangle \ || \ \text{IN}(r(e2))$
 $stk = \text{EMPTY} \ \rightarrow g(p, stk, vs) = vs \ || \ \text{IN}(p)$

respectively. The first 2 of these functions imply the following behavior for an arbitrary stack where vs has the value $\langle \rangle$:

$stk = (e1, \dots, en) \ \& \ vs = \langle \rangle \ \rightarrow g(p, stk, vs) =$
 $\text{IN}(p) \ || \ (\langle e1 \rangle \ || \ \text{IN}(e1) \ || \ \dots \ || \ \langle en \rangle \ || \ \text{IN}(en))$

and in combination with the last function, the general behavior

$stk = (e1, \dots, en) \ \rightarrow g(p, stk, vs) =$
 $vs \ || \ \text{IN}(p) \ || \ (\langle e1 \rangle \ || \ \text{IN}(e1) \ || \ \dots \ || \ \langle en \rangle \ || \ \text{IN}(en))$

is suggested.

In Example 6, we saw that the technique generalizes to the use of three (and indeed an arbitrary number of) constraints. We have seen that each of these represents a subset of the loop function g being sought. If the constraints themselves are sufficiently general, it may be that the first several of these, taken collectively, constitute a complete description of g . We consider this situation in the following section.

5. Complete Constraints

The technique described above for obtaining a general loop function is "nondeterministic" in that the constraints 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 constraints. 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 - For some $N > 0$, an initialized loop is N-closed with respect to its specification f iff the disjunction of the constraints C_1, C_2, \dots, C_N defines a value of the function g over a set for which the loop is closed. In this case, the constraints C_1, C_2, \dots, C_N are complete.

Thus if a loop is N -closed for some $N > 0$, the disjunction of the first N constraints 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 7 - Consider the following program

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

The first constraint is

$C_1: b_0 \geq 0 \ \& \ a = a_0 + 1 \ \& \ b = b_0 \ \rightarrow \ g(a, b) = a_0 + b_0 + 1$

which SIMPLIFIES to

$b \geq 0 \ \rightarrow \ g(a, b) = a + b.$

This constraint defines a value of g over the set of data states in which b is non-negative. Since the loop is closed for this set (i.e., if b is initially non-negative, it remains so after each WHILE loop iteration), the program is 1-closed. Thus C_1 by itself constitutes an adequate loop function.

Initialized loops that 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 constraints) is deterministic.

Example 8a - Consider the program

```

{seq=seq0}
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 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 constraint defines g over all possible values of sum and seq and the loop is trivially closed for this set.

Example 8b - As a second illustration of a 2-closed initialized loop, the following program tests whether a particular key appears in an binary search tree.

```

{tree=tree0}
success := FALSE;
while tree ≠ NIL & ~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) = 'key occurs in binary search tree tree0'. This program is also 2-closed. Note that the first constraint

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 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).

The disjunction of these two constraints defines a value of g over the set

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

The loop is closed for this set and hence the initialized loop is 2-closed.

Example 8c - Consider the sequence of initialized loops P1,P2,P3 ... defined as follows for each I>0:

PI : {y=y0 & x=x0 & x0>=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 constraints for program PI are

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

where MI(x) = 'x is a multiple of I'. Since the disjunction of these is the constraint

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

which defines a value of g over a set for which the loop is closed, we conclude that for each I>0, program PI is I-closed.

For many initialized loops that seem to occur in practice, however, there does not exist an N such that they are N -closed with respect to their specification. This means that no finite number of constraints 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 1. The constraints C_1, C_2, C_3, \dots define the general loop behavior for $z=0, z=k, z=2*k, \dots$ 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 covered by the first N constraints.

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 states that occur after an arbitrary number of iterations are indistinguishable from states that occur after zero (or one) loop iterations. To illustrate, consider the 2-closed initialized loop of Example 8a that 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 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 that 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 that was "inconsistent" across all values of the loop inputs and which could only be inferred from the constraints 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 constraints.

6. 'Tricky' Programs

The above heuristic suggests inferring g from two constraints on that function, $C1$ and $C2$. Constraint $C2$ is of particular importance since REWRITE and SUBSTITUTE are applied to $C2$, and, consequently, it 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 representative of the unknown 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 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 9 - Consider applying the technique to the following program which is an alternative implementation of the integer multiplication program presented in Example 1:

```
{v=v0 & v0>=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 constraints $C1$ and $C2$ are identical to those for the program in Example 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 that 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 9.

7. 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. Our technique sacrifices determinism in favor of wide applicability and ease of use. It handles in a fairly straightforward manner typical programs in these two classes (e.g. Examples 1-3) as well as a number of programs that do not fit in either of the classes (e.g. Examples 4-6).

Due to the close relationship between loop functions and loop invariants (see, for example, [Morris & Wegbreit 77]), 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 2. The loop exit condition can be used to obtain an initial loop invariant approximation

```
d=0 -> w=c0**d0.
```

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

```
(d=0 -> w=c0**d0) & (d=1 -> w*c=c0**d0).
```

In the analysis presented in Example 2, we obtained a value for the

general function for each of two different values of the initialized variable w (i.e. 1 and $\text{SQRT}(c)$); here we have obtained a "value" for the loop invariant we are seeking for each of two different values of the variable that controls the termination of the loop d . Applying the analysis in [Morris & Wegbreit 77], these loop invariant "values" can be translated to constraints as follows:

$$\begin{aligned} d=0 &\rightarrow g(w,c,d)=w, \\ d=1 &\rightarrow g(w,c,d)=w*c. \end{aligned}$$

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 obtained in Example 2.

We summarize the relationship between these two techniques as follows. As the initialized loop in question operates on some particular input, let $X[0], X[1], \dots, X[N]$ 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 paper 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 and the program to compute Ackermann's function 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 constraints. In the other technique, this information must somehow be inferred from the corresponding constraints (e.g. by looking for a pattern, etc.). This difficulty is inherent in any "bottom up" approach to synthesizing g .

8. Concluding Remarks

In this paper we have proposed a technique for deriving functions that 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 that must be read, understood, modified and verified by "maintenance" personnel. We offer the heuristic as a tool that 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 5, this idea is applied to initialized loop programs. The result is that a loop function g for a loop that is N -closed (for some $N > 0$) can be synthesized in a deterministic manner by considering the first N constraints. 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 that are not "tricky" (as discussed in Section 6). Preliminary results along this line are described in [Dunlop & Basili 81] (see also [Basu 80]).

9. Acknowledgement

The authors are extremely indebted to D. Gries and B. Witt for very lengthy and constructive reviews of an earlier draft of this report.

10. References

[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.

[Dunlop & Basili 81]

Dunlop, D. and Basili, V. Generalizing Specifications for Uniformly Implemented Loops, University of Maryland Computer Science Technical Report TR-1116, October 1981.

[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.

[Gries 79]

Gries, D. Is Sometime Ever Better Than Always?, 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.

[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 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, University 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.

[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.