A Heuristic for Deriving Loop Functions

DOUGLAS D. DUNLOP AND VICTOR R. BASILI, MEMBER, IEEE

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.**

Index Terms—**Constraints, initialized loop programs, loop functions, program verification.**

I. INTRODUCTION

IN this paper, 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. 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 synthesize heuristically a sufficiently strong inductive assertion (i.e., loop invariant [4], [8], [9]) 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, [11], [18]). 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 a hypothesis concerning the function of the WHILE loop. Once this has been done, the loop can be proven/disproven correct with respect to the hypothesis using standard techniques [2], [13]-[15], [17], [19]. If the hypothesis is shown to be valid, the correctness/incorrectness of the program in question follows immediately. It has been shown [1], [3], [15], [16] 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 is exploited in order to obtain the hypothesis. Our approach is not deterministic in general, but is intended to be more widely applicable and easier to use than those previously proposed.

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. One might proceed by "executing" the loop by hand on several sample inputs and then inferring some general expression based on these results. Decisions that need to be made when using such a technique include how many sample inputs to use, how to select inputs, and how to infer the general expression. But hand execution can be a difficult and an error prone task. Indeed, the loops for which hand execution can be carried out straightforwardly are often the ones 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.

Manuscript received January 21, 1982, revised March 28, 1983 and September 22, 1983. This work was supported in part by the U.S. Air Force Office of Scientific Research under Contract AFOSR-F49620-80-C-001.

D. D. Dunlop was with the Department of Computer Science, University of Maryland, College Park, MD 20742. He is now with Intermetrics, Inc., 4733 Bethesda Avenue, Bethesda, MD 20814.

V. R. Basili is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

0098-5589/84/0500-275$01.00

II. Constraints 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 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* [2], [15] 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 [6], [10]. 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 a 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 versus just code). The disadvantage 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

$$X0 \in D(f) \rightarrow f(X0)=g(K(X0)). \tag{1}$$

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" [15]

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

of the standard technique for showing the loop computes g must hold [14], [15], [17]. This implies

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

Combining with (1) yields

$$X0 \in D(f) \,\&\, B(K(X0)) \rightarrow f(X0)=g(H(K(X0))). \tag{2}$$

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

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

and

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

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. Constraint (1′) views the program as initialization followed by the loop—as a single entity. Constraint (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

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

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 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 $v0 \geqslant 0$:

```
{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: $v=v0 \;\&\; v0 \geqslant 0 \;\&\; z=0 \rightarrow g(z,v,k)=v0*k$
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 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 v0 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 v0 for that constraint. We therefore rewrite C1 and C2 as

C1: $v \geqslant 0 \;\&\; z=0 \rightarrow g(z,v,k)=v*k$
C2: $v \geqslant 0 \;\&\; z=k \rightarrow g(z,v,k)=(v+1)*k.$

Next, note that argument z of g 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 these function expressions. First, rewrite (v + 1) * k as v * k + k:

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

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

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

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 g. With it, we can prove the above program using the standard techniques [14], [15].

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 function g from the constraints. This method was just used on the previous example, and we will illustrate its use on many other examples in this paper.

III. 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 ∈ 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: $v0 \geqslant 0 \;\&\; v=v0 \;\&\; z=0 \rightarrow g(z,v,k)=v0*k$
C2: $v0>0 \;\&\; v=v0-1 \;\&\; z=k \rightarrow g(z,v,k)=v0*k.$

We make the following comments. First, g is 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 ∈ D(f) (i.e., v0 ⩾ 0) and B(K(X0)) (i.e., v0 ≠ 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: $v \geqslant 0 \;\&\; z=0 \rightarrow g(z,v,k)=v*k.$

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

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

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. Variable z is a candidate to be introduced into the function expression (v + 1) * k. It is equal to 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: $v \geqslant 0 \;\&\; z=k \rightarrow g(z,v,k)=v*k+k.$

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 = 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:

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

and

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

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 four steps to obtain a hypothesis

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

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 [14]. Specifically, the hypothesis is valid if and only if each of

- the loop terminates for all $v \geqslant 0$,
- $v = 0 \rightarrow z = v * k + z$, and
- $v * k + z$ is a loop constant (i.e., $v0 * k0 + z0 = v * k + z$ 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: $v \geqslant 0 \;\&\; z = k \rightarrow g(z,v,k) =$
$v*k + 3*k + k*k*(v-7)/(4*k) + k*k*k/(k*k)$
$- k*k*k*(v-7)/(4*k*k) - k*k*k*3/(k*k)$

and

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

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 for many initialized loops 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.

IV. 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:

$$d0 \geqslant 0 \;\&\; c = c0 \;\&\; d = d0 \;\&\; w = 1 \rightarrow g(w,c,d) = c0**d0$$

and SIMPLIFIES to

$$C1:\; d \geqslant 0 \;\&\; w = 1 \rightarrow g(w,c,d) = c**d.$$

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

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

which SIMPLIFIES to

$$C2a:\; d \geqslant 0 \;\&\; c = w*w \rightarrow g(w,c,d) = w**(d*2+1).$$

The constraint corresponding to the other loop-body path is

$$d0 > 0 \;\&\; \neg odd(d0) \;\&\; w = 1 \;\&\; c = c0*c0 \;\&\; d = d0/2 \rightarrow g(w,c,d) = c0**d0$$

and SIMPLIFIES to

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

i.e.,

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

where SQUARE(x) = "x is a perfect square," and SQRT(x) = "the square root of the perfect square x." The term 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

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

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, in our experience, 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 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. Predicate empty(x) is TRUE if x is the empty tree:

```
{¬empty(t)}
n := 0; s := {t};
while s ≠ {} do
    select and remove some element e from s;
    n := n + 1;
    /* s := s ∪ SONS(e) */
    if ¬empty(left(e)) then s := s ∪ {left(e)} fi;
    if ¬empty(right(e)) then s := s ∪ {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: ¬empty(t) & $n=0$ & $s=\{t\} \rightarrow g(n,s)=$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

$s := s \cup$ SONS(e)

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

C2: ¬empty(t) & $n=1$ & $s=$SONS(t) $\rightarrow g(n,s)=$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 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 $A \in B$. Applying SUBSTITUTE in the obvious way yields

¬empty(t) $\rightarrow g(n,s)=n+$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 condition ¬empty(t), which 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 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 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

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

only under the provision that 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 [7]: 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 $\langle s(n), \cdots, s(2), s(1)\rangle$.

```
{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 three paths through the loop body are

C2a: $m=0 \;\&\; n\geq 0 \;\&\; s=\langle n+1\rangle \rightarrow g(s)=\langle A(m,n)\rangle$
C2b: $m>0 \;\&\; n=0 \;\&\; s=\langle m-1,1\rangle \rightarrow g(s)=\langle A(m,n)\rangle$
C2c: $m>0 \;\&\; n>0 \;\&\; s=\langle m-1,m,n-1\rangle \rightarrow g(s)=\langle A(m,n)\rangle$.

REWRITING these three based on the above definition of A yields

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

SUBSTITUTING here yields

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

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

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

We remark that in the first three 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, the verifier is left to infer a behavior for a sequence of arbitrary length.

Example 5: Let v[1 : n], n > 0, contain 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: $m=0 \;\&\; i=1 \rightarrow g(m,i,v,n)=\text{AMAX}(v)$
C2: $m=v[1] \;\&\; i=2 \rightarrow g(m,i,v,n)=\text{AMAX}(v)$.

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,

$$g(m,i,v,n)=\text{MAX}(m,\text{AMAX}(v[i:n])),$$

agrees with C1.

Example 6: Let p be a pointer to a node in a binary tree and IN(p) be the sequence of pointers that point to the nodes in an in-order 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. 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 := push(stk,p) /* push p onto stk */ ;
      p := l(p)
   else
      p := top(stk) /* p gets top stk element */ ;
      stk := pop(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. Constraints 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 (e1, ··· , en) for a stack containing the elements e1, ··· , en 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. Functions C3a and C3b correspond to executions of the first and second loop-body paths (on the second iteration), respectively. There is only one 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≠NIL & p=l(rt) & stk=(rt) & vs=<> →
g(p,stk,vs)= IN (l(rt)) ‖<rt>‖ IN (r(rt))

C3a′: rt≠NIL & l(rt)≠NIL & p=l(l(rt)) & stk=(l(rt),rt) & vs=<> →
g(p,stk,vs)= IN (l(l(rt))) ‖<l(rt)>‖ IN (r(l(rt)))
‖<rt>‖ IN (r(rt))

C3b′: rt≠NIL & l(rt)= NIL & p=r(rt) & stk=EMPTY & vs=<rt>→
g(p,stk,vs)= <rt> ‖IN (r(rt)).

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

stk=(e1) & vs=<> → g(p,stk,vs)=IN (p) ‖<e1>‖ IN (r(e1))
stk=(e1,e2) & vs=<> → g(p,stk,vs)=IN (p) ‖<e1>‖ IN (r(e1))
‖<e2>‖ IN (r(e2))
stk=EMPTY → g(p,stk,vs)=vs ‖IN (p),

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

stk=(e1, ··· , en) & vs=<> → g(p,stk,vs) =
IN (p) ‖ (<e1> ‖ IN (e1) ‖ ··· ‖ <en> ‖ IN (en))

and in combination with the last constraints, the general behavior

stk=(e1, ··· , en) → g(p,stk,vs) =
vs ‖ IN (p) ‖ (<e1> ‖ IN (e1) ‖ ··· ‖ <en> ‖ IN (en)).

V. 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 to fill in the remaining "pieces" of the loop function "picture" once this basis has been established.

In some circumstances, however, the constraints do constitute a complete description of an adequate loop function. The significance of these situations is that no guessing 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 if the disjunction of the constraints C1, C2, ··· , CN defines a value of function g over a set for which the loop is closed. In this case, the constraints C1, C2, ··· , CN 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, 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

C1: b0≥0 & a=a0+1 & b=b0 → g(a,b)=a0+b0+1

which SIMPLIFIES to

b≥0 → g(a,b)=a+b.

This constraint defines a value of g over the set of data states in which b is nonnegative. Since the loop is closed for this set (i.e., if b is initially nonnegative, it remains so after each itera-

tion), the program is 1-closed. C1, by itself, constitutes an adequate loop function.

Initialized, 1-closed loops 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 a binary search tree:

```
{tree=tree0}
success := FALSE;
while tree ≠ NIL & ¬success do
   if    name(tree) = key then success := TRUE
   lseif 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. 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

{<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 1 > 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, program PI is I-closed.

Many initialized loops are not N-closed with respect to their specification for any N: no finite number of constraints will pinpoint the appropriate generalization exactly. Thus, 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 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 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 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 because 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. After an arbitrary number of iterations, little can be said about the history of execution based on the values of sum and seq. All one can say is that if sum is not zero then at least 1 loop iteration was executed, but the number of these iterations may be 1, 10, or 10 000.

In contrast, again consider the integer multiplication program of Example 1, an initialized loop we know not to be N-closed for any N. Based on the values of the program variables z, v, and k after some number of iterations, what can be said about the history of the execution? Well, exactly z/k iterations have occurred and we can reconstruct each previous value of 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 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 that 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 that allows their general behavior to be described completely by the first one or two constraints.

VI. "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 aribitrary 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 unexpected 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 "unexpected" 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}.
```

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, if 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 not typical of its general behavior, due to the high dependence of the loop-body behavior on the input value of 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 that 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.

VII. Related Work

In [3], [15], [16] 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, [17]), 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 [11], [18]. 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.

For example, consider the exponentiation program of Example 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 2, we obtained a value for the general function 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 that controls the termination of the loop d. Applying the analysis in [17], these loop invariant "values" can be translated to constraints as follows:

$$d=0 \rightarrow g(w,c,d)=w,$$
$$d=1 \rightarrow g(w,c,d)=w*c.$$

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 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], $\cdots$, 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 is 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 and 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 state carefully the relative merits of these two opposing techniques. In our view, however, in a number of circumstances 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.

VIII. 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 [14] or subgoal induction [17] 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. However, a large number of existing programs must be read, understood, modified and verified by "maintenance" personnel. We offer the heuristic as a tool intended to facilitate these tasks.

It has been argued [15] that the notion of closure of a loop with respect to an input domain is fundamental in analyzing the loop. In Section V, this idea is applied to initialized loop programs. The result is that a loop function 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 VI). Preliminary results along this line are described in [5] (see also [1]).

Acknowledgment

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

References

[1] S. Basu, "A note on synthesis of inductive assertions," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 32-39, Jan. 1980.
[2] S. Basu and J. Misra, "Proving loop programs," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 76-86, Mar. 1975.
[3] —, "Some classes of naturally provable programs," in *Proc. 2nd Int. Conf. Software Eng.*, San Francisco, CA, Oct. 1976, pp. 400-406.
[4] E. W. Dijkstra, *A Discipline of Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1976.
[5] D. Dunlop and V. Basili, "Generalizing specifications for uniformly implemented loops," Univ. Maryland, Comput. Sci. Tech. Rep. TR-1116, Oct. 1981.
[6] H. Ellozy, "The determination of loop invariants for programs with arrays," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 197-206, Mar. 1981.
[7] D. Gries, "Is sometime ever better than always?," *Trans. Programming Languages and Syst.*, vol. 1, pp. 258-265, Oct. 1979.
[8] —, *The Science of Programming.* New York: Spring-Verlag, 1981.
[9] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, pp. 576-583, Oct. 1969.
[10] S. Katz and Z. Manna, "A heuristic approach to program verification," in *Proc. 3rd Int. Joint Conf. Artificial Intell.*, Stanford, CA, 1973, pp. 500-512.
[11] —, "Logical analysis of programs," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 188-206, Apr. 1976.
[12] Z. Manna and R. Waldinger, "Towards automatic program synthesis," Stanford Artificial Intell. Project, Memo. AIM-127, July 1970.
[13] H. D. Mills, "Mathematical foundations for structured programming," IBM Federal Syst. Div., FSC 72-6012, 1972.

[14] —, "The new math of computer programming," *Commun. Ass. Comput. Mach.*, vol. 18, pp. 43-48, Jan. 1975.
[15] J. Misra, "Some aspects of the verification of loop computations," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 478-486, Nov. 1978.
[16] —, "Systematic verification of simple loops," Univ. Texas, Tech. Rep. TR-97, Mar. 1979.
[17] J. H. Morris and B. Wegbreit, "Subgoal induction," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 209-222, Apr. 1977.
[18] B. Wegbreit, "The synthesis of loop predicates," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 102-112, Feb. 1974.
[19] —, "Complexity of synthesizing inductive assertions," *J. Ass. Comput. Mach.*, vol. 24, pp. 504-512, July 1977.

Douglas D. Dunlop received the B.S. degree in mathematics and the M.S. degree in computer science from the College of William and Mary, Williamsburg, VA, in 1975 and 1977, respectively, and the Ph.D. degree in computer science from the University of Maryland, College Park, in 1982.

He is currently a Software Engineer for Intermetrics, Inc., Bethesda, MD. He joined Intermetrics in 1982, working on studies relating to the transistion of existing military computer systems to Ada and its associated software technology. He is currently involved in the development of a software support environment for the analysis and verification of VHSIC Hardware Description Language (VHDL) designs.

Dr. Dunlop is a member of the Association for Computing Machinery.

Victor R. Basili (M'83) is Professor and Chairman of the Department of Computer Science at the University of Maryland, College Park. He was involved in the design and development of several software projects, including the SIMPL family of programming languages. He is currently measuring and evaluating software development in industrial settings and has consulted with many agencies and organizations, including IBM, GE, CSC, Naval Research Laboratory, Naval Surface Weapons Center, and NASA. He has authored over 50 published papers on the methodology, the quantitative analysis, and the evaluation of the software development process and product. In 1982, he received the Outstanding Paper Award from the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. He was Program Chairman for the 6th International Conference on Software Engineering, and the first ACM SIGSOFT Software Engineering Symposium on Tools and Methodology Evaluation. He serves on the Editorial Boards of the *Journal of Systems and Software* and the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING.

Dr. Basili is a member of the Association for Computing Machinery, the Executive Committee of the Technical Committee on Software Engineering, and the IEEE Computer Society.

0098-5589/84/0500-285$01.00 © 1984 IEEE