A Comparison of the Axiomatic and Functional Models of Structured Programming

VICTOR R. BASILI AND ROBERT E. NOONAN

***Abstract*—This paper discusses axiomatic and functional models of the semantics of structured programming. The models are presented together with their respective methodologies for proving program correctness and for deriving correct programs. Examples using these methodologies are given. Finally, the models are compared and contrasted.**

***Index Terms*—Axiomatic correctness, functional correctness, program derivation, structured programming.**

I. Introduction

STRUCTURED programming methodology has centered around the use of a particular set of program constructs, chosen because they provide the programmer with the ability to maintain control of the program development. This control involves the ability to break the program into small easily understood pieces using a few simple structures which permit the verification of each step in the development process before going on to the next step. Each step involves the decomposition of the current set of pieces into another set of simple structures. This development technique is commonly referred to as stepwise refinement.

The guidelines for choosing these basic structures and the ability to prove the correctness of the partial solution of a program are based on formal models of the semantics of various program constructs. Associated with each of these constructs is a rule for verifying the correctness of the expansion from its specification or intended function. This paper discusses an axiomatic model (Floyd [3], Hoare [6]) and a functional model (Mills [8], [9]).

It should be noted that these models were originally defined in different frameworks: Floyd's with program flowcharts, Hoare's model with programming language statements, and Mills' model with single-entry-single-exit program segments. For the purposes of comparison, variants of the basic Hoare and Mills models have been chosen and defined within the common framework of the standard structured programming constructs. For those readers familiar with one model or the other, the discussion of the specific correctness and derivation techniques may be glossed over, but they should be read in order to provide a common framework for comparison.

In the next section, the basic models will be given along with the methodology used for proving the correctness of a program in each model. Section III gives the algorithm for deriving a correct program and contains an example derivation using each model. Section IV compares the two models: their similarities, differences, and interrelationships.

Manuscript received June 23, 1978; revised March 20, 1979. This work was supported in part by the U.S. Air Force Office of Scientific Research under Grant AFOSR77-31 81 to the University of Maryland.

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

R. E. Noonan is with the Department of Mathematics and Computer Science, College of William and Mary, Williamsburg, VA 23185.

II. Basic Models of Program Constructs and Correctness

In this section the variants of the Hoare and Mills models are defined. Although the Hoare model is oriented toward programming language statements and the Mills model toward single-entry-single-exit program constructs, by restricting the constructs used to the standard structured programming ones, both models can be defined over precisely the same constructs. This is done to facilitate a comparison of the two models. For the remainder of this paper, the terms *statement* and *program construct* will be used interchangeably.

The semantic rules are given below for each program construct, where S1 and S2 are again constructs from this set:

```
1) assignment:  x := f
2) sequence:    S1 ; S2
3) iteration:   while b do S1 od
4) choice:      if b then S1 else S2 fi
                  if b then S1 fi.
```

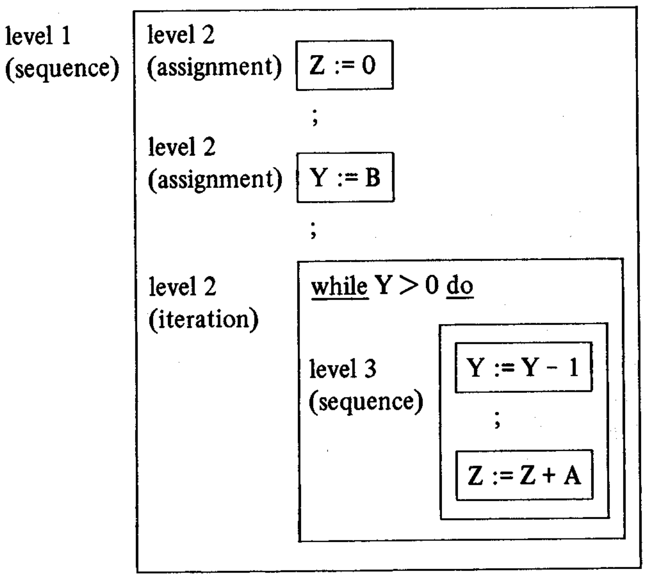
For the purpose of this paper, a program consisting of only these constructs will be called a *structured program*. These particular constructs were chosen because they are representative, they are the most commonly used, and they are sufficient for the examples given in this paper.

Based on the semantic definitions of the individual constructs, we show the standard correctness criteria for each construct and how to apply the correctness criteria to a structured program. These program constructs permit breaking the program into a hierarchy of structured subprograms such that the correctness of each subprogram can be proved independent of the rest of the program. Given any of the allowable program constructs discussed earlier, we can create this program hierarchy by defining each of those statements to be at a particular level in the hierarchy. Then consider the constructs S1 and S2 used in the definition of those basic constructs to be single logical statements at the next lower level. This creates a hierarchy in which the lowest level must consist solely of assignment statements.

As an example, consider the program given in Fig. 1. The program hierarchy is the following:

```
1.   Z  :=  0 ;

2.   Y  :=  B ;

3.   while  Y  >  0   do

4.          Y :=  Y - 1 ;

5.          Z :=  Z + A

6.   od
```

Fig. 1. Simple multiplication program.



A. Axiomatic Correctness

The particular model of axiomatic correctness that we use here is due to Hoare [6].

> The intended function of a program, or part of a program can be specified by making general assertions about the values which the relevant variables will take *after* execution of the program. These assertions will usually not ascribe particular values to each variable, but will rather specify certain general properties of the values and the relationships holding between them. . . .In many cases, the validity of the results of a program (or part of a program) will depend on the values taken by the variables before that program is initiated. These initial preconditions of successful use can be specified by the same type of general assertion as is used to describe the results on termination.

The notation

{P} S {Q}

is used to state the required connection between the input assertion P, output assertion Q, and program (or part of a program) S. The program S is *partially correct* with respect to P and Q iff for every substitution of values which makes P true, then after the execution of S, Q must be true. To prove (*total*) *correctness*, we must also prove that if P is true then S terminates.

```
Assignment

A1  {P(x⇐f)}  x := f  {P}
      where P(x⇐f) is obtained from  P  by substituting
      f  for all free occurrences of  x.

Composition

A2   If  {P}  S1  {R}  and  {R}  S2  {Q},
          then  {P}  S1;  S2  {Q}.

Iteration

A3   If  {P, B}   S  {P},
          then  {P}  while  B  do  S  od  {P,¬B}

Choice

A4   If  {P, B}   S1   {Q}  and  {P,¬B}   S2 {Q},
          then  {P}  if B then S1 else S2 fi {Q}

A5   If  {P, B}   S1   {Q}  and   P,¬B → Q,
          then  {P}  if  B  then  S1  fi  {Q}

Consequence

A6   If  {P}  S  {Q}  and Q → R,  then  {P}  S  {R}.

A7   If  {Q}  S  {R}  and P → Q,  then  {P}  S  {R}.
```

Fig. 2. Hoare's rules of inference.

In this paper, we ignore the problem of termination since the techniques used are identical for both models. For most "real-world" programs, the method of proving termination by using well-founded sets due to Floyd [3] has proved adequate. All of the example programs in this paper have trivial proofs of termination.

To prove that S is correct with respect to P, Q, a first-order predicate theory is used together with an axiom scheme or rule of inference for each program construct. These latter are given in Fig. 2. A complete explanation of these can be found in Hoare [6]. Note that the notation P, Q is used to denote the formula $P \wedge Q$. In the iteration rule A3 the assertion P is usually called the *loop invariant*.

To prove the correctness of a particular program, assume that every program statement, as well as the program itself, has both a pre- and postassertion and the program hierarchy has been produced. Since each statement in the hierarchy has both a pre- and postassertion, the correctness of each piece is proved using the rules given in Fig. 2 based on the form of the program hierarchy. This demonstrates the partial correctness of the program.

In practice, however, every statement in a program is rarely tagged with both its pre- and postconditions. The minimum effective requirement is to be given the pre- and postconditions for the entire program and the loop invariant for each loop. In this case, the pre- and postconditions can be generated by a top down process.

To produce the assertions, first produce the program hierarchy using Algorithm A. In this case, a slight modification (or normalization) of the process is introduced; given a sequence S1; · · · ; Sn - 1 ; Sn consider it as a composition

(S1; · · · ; Sn - 1) ; Sn. Given the hierarchy, the precondition of each statement is generated by a top down recursive algorithm from the postcondition (note that the output assertion of the entire program is assumed to be given).

Algorithm A:

1) *Assignment:* x := f. If the post condition is P, then the precondition is P(x ⇐ f) as given in rule A1 of Fig. 2.

2) *Sequence:* S1 ; S2. If the post condition is Q, then the process is invoked recursively to find the precondition of S2, namely, R. Repeat algorithm A on S1 given R as its postcondition.

3) *Iteration:* while B do S od. Note that it is assumed that the loop invariant P is given. If Q is the postcondition, it must be proved (as part of the proof process) that $P, \neg B \rightarrow Q$. P is the precondition of the loop. {P, B} is the given precondition of S and P the postcondition of S. The process of finding preconditions is continued by invoking Algorithm A on S.

4) *Choice:* if B then S1 else S2 fi. If Q is the postcondition of the if, then the process is repeated by invoking Algorithm A on S1 and on S2 with Q as their postconditions, yielding P1 and P2 as their respective preconditions. The precondition of the if is $\{(B \rightarrow P1), (\neg B \rightarrow P2)\}$. Note that this condition can normally be greatly simplified. For example, if P1 is of the form {P, B} and P2 is of the form $\{P, \neg B\}$ then the precondition is P.

The proof can then be carried out as before, but much of the work has been eliminated. Since the preconditions are generated from the postcondition, the associated program part must be correct with respect to these conditions. Only when a precondition is given (e.g., a loop invariant) must it be proved that the desired precondition is implied by the given precondition.

In order to illustrate this method, the correctness of the program given in Fig. 3 is proven. This program computes the product of two natural numbers A and B by repeated addition. Note that in order to be more precise the output assertion should state that neither A nor B is modified by the program; in the axiomatic model it is assumed that unless otherwise stated no program modifies its input values (this restriction is important, as shall be seen later).

To demonstrate that the program is partially correct (i.e., that it does compute A * B), a more formal argument than usual is given; this formality is dropped in later sections. Strictly speaking, the proof given in Fig. 4 is a derivation of a proof, since a proof is normally bottom-up. In Fig. 4 line numbers are used to refer to program parts of more than one statement.

```
   { A ≥ 0 , B ≥ 0 }

1.   Z := 0 ;

2.   Y := B ;


   { Z = A * (B - Y) , Y ≥ 0 }

3.   while Y > 0 do

4.      Y := Y - 1 ;

5.      Z := Z + A

      od

   { Z = A * B }
```

Fig. 3. Axiomatic correctness of simple multiplication.

B. Functional Correctness

The model for functional correctness was developed by Mills [8], [9]. Here the intended function of a program is stated as a functional abstraction which summarizes the possible outcomes of the program part under consideration independent of the internal control structure and data operations. The goal is to produce loop-free, branch-free, and sequence-free descriptions of the effects of programs on data. The question of correctness reduces to the question of functional equivalence between the program under consideration and the control-free functional version of the program with respect to its intentional effect on data, i.e., the high level function abstracts out any local variables, etc. This functional abstraction is meant to represent a function in the strict mathematical sense, i.e., given an input vector I and an output vector O, the function (program segment) defines a rule for finding O given I, namely, O = f(I).

The program S is *partially correct* with respect to f iff for every argument X such that f is defined and f(X) = Y, then if program S is executed with initial state vector X, its final state vector is Y. To prove *(total) correctness*, we must prove that if X is an element of the domain of f then S terminates. (In this paper, we ignore the problem of proving termination since the techniques used are identical for both models.)

To prove that S is correct with respect to f, function composition and equivalence are used together with functional definitions for each program construct. These functional definitions and equivalences are given in Fig. 5. The square bracket notation is used to denote the function represented by the program construct contained inside the brackets, i.e., [S] represents the function computed by the statement S.

Note that the rule for assignment given in F1 is a simple function. A function is a model of assignment in that it maps input values into output values; however, it differs from an assignment in that the input and output states are always distinguishable. It is convenient to write the intended function of a program segment as a generalized assignment, in which the subscript "in" is attached to all input variables in order to emphasize this separation of value spaces. From the point of view of an assignment statement, it is as though there is a distinct copy of the input values.

As in the case of the axiomatic model, in order to prove correctness of a particular program, assume that every statement S in the hierarchy has an intended function [S] associated with it. Then merely show the equivalence of each statement to its intended function using the rules given in Fig. 5. This yields the proof of partial correctness of the program. In order to more clearly identify statements with their associated intended functions, the parts of the statement will be numbered sequentially as a subunit of the intended function, e.g., 3.1, 3.2, etc., for a function numbered 3.

In practice, however, every statement is rarely tagged with

Step Number	Proof	Derived from Step	Justification
1	$\{A \geq 0 \quad B \geq 0\}$ lines 1 - 5 $\{Z = A * B\}$	--	Original problem
2	$\{A \geq 0 \quad B \geq 0\}$ lines 1 - 2 $\{Z = A *(B-Y) \quad Y \geq 0\}$	1	Rule A2
3	$\{Z = A *(B-Y) \quad Y \geq 0\}$ lines 3 - 5 $\{Z = A * B\}$	1	Rule A2
4	$\{Z = A *(B-B) \quad B \geq 0\}$ Y := B $\{Z = A *(B-Y),\ Y \geq 0\}$	2	Rule A2; truth follows from A1
5	$\{A \geq 0 \quad B \geq 0\}$ Z := 0 $\{Z = A *(B-B) \quad B \geq 0\}$	2, 4	Rule A2
6	$\{0 = A *(B-B) \quad B \geq 0\}$ Z := 0 $\{Z = A *(B-B) \quad B \geq 0\}$	5	Rule A7, since $\{A \geq 0,\ B \geq 0\} \rightarrow \{0 = A *(B-B),\ B \geq 0\}$; truth follows from A1
7	$\{Z = A *(B-Y) \quad Y \geq 0\}$ lines 3 - 5 $\{Z = A *(B-Y) \quad Y \geq 0,\ Y \leq 0\}$	3	Rule A6, since $\{Z = A *(B-Y),\ Y \geq 0,\ Y \leq 0\} \rightarrow \{Z = A * B\}$
8	$\{Z = A *(B-Y) \quad Y > 0\}$ lines 4-5 $\{Z = A *(B-Y) \quad Y \geq 0\}$	7	Rule A3
9	$\{Z + A = A *(B-Y) \quad Y \geq 0\}$ Z := Z + A $\{Z = A *(B-Y) \quad Y \geq 0\}$	8	Rule A2 truth follows from A1
10	$\{Z = A *(B-Y),\ Y > 0\}$ Y := Y - 1 $\{Z + A = A *(B-Y),\ Y \geq 0\}$	8, 9	Rule A2
11	$\{Z + A = A *(B - (Y-1)),\ Y - 1 \geq 0\}$ Y := Y - 1 $\{Z + A = A *(B-Y),\ Y \geq 0\}$	10	Rule A7, since $\{Z = A * (B-Y),\ Y > 0\} \rightarrow \{Z + A = A *(B-(Y-1)),\ Y - 1 \geq 0\}$; truth follows from A1

Fig. 4. Derivation of an axiomatic proof of simple multiplication.

Assignment

F1 $f(x)$ (where the assignment is $x := f(x)$)

Composition

F2 $f(x) = [\,g\,;\,h\,](x) = h(g(x))$

Iteration

F3 $f(x) = [$while p do g od$](x)$

$$= \begin{cases} f(g(x)) & \text{if } p(x) \\ x & \text{if } \neg p(x) \end{cases}$$

Choice

F4 $f(x) = [$if p then g else h fi$](x)$

$$= \begin{cases} g(x) & \text{if } p(x) \\ h(x) & \text{if } \neg p(x) \end{cases}$$

F5 $f(x) = [$if p then g fi$](x)$

$$= \begin{cases} g(x) & \text{if } p(x) \\ x & \text{if } \neg p(x) \end{cases}$$

Fig. 5. Functions computed by program constructs.

its intended function. The minimum effective requirement is to be given the intended function for the entire program and for each loop. In this case, the proof is carried out by showing the functional equivalence of each given function [S] and its associated statement S at each level in the hierarchy of the program. In this process, the functions computed at the various levels in the hierarchy may be effectively created using a top-down recursive trace or symbolic execution (Hantler [5]) algorithm through the levels of the program as follows.

Algorithm B:

1) *Assignment:* The value of the left-hand side variable is symbolically replaced by the function computed by the right-hand side.

2) *Sequence:* Trace through the first statement followed by the second statement.

3) *Iteration:* Use the given intended function in the trace.

4) *Choice:* Split the trace process into two cases, the *then* part and the *else* part, and continue the trace process through each part separately.

In order to illustrate this method, the partial correctness of the program given in Fig. 6 is demonstrated in Fig. 7. Note that this program is nearly identical to the one given in Fig. 3, but does not save the value of the variable B. (Note: the algorithm could have been written exactly as was done for the axiomatic model example, but was not to demonstrate a point in Section IV.)

III. Formal Program Derivation

In this section, an algorithm for carrying out a derivation of a correct program is given and an example presented for each model.

A. Algorithms for Formal Derivation

The algorithms used for carrying out a formal derivation are surprisingly similar in the two models. In each case, the stepwise refinement methodology is applied leading usually to new subproblems to be solved and lemmas to be proved. The primary difference between the two methods is in stating the problem requirements of the problem at each level of the decomposition process.

Using the lemmas and subproblems shown in Table I, the algorithm for carrying out an axiomatic derivation can be stated as follows (note sets are denoted as [[· · ·]]).

Algorithm C:

```
/* Given the specifications, organize them into a function f
   to be computed. Let P be the relations (assertions)
   necessary to restrict the input domain of f. Let Q be
   the assertion which "captures" the output of f. */
problems-to-be solved := [[ {P} S {Q}, for unknown S]]
while problems-to-be solved ≠ empty do
   "using some strategy, choose a specific element of problems-
      to-be-solved, denoted {P} S {Q};
   "choose an appropriate program construct for S and in the
      case of sequence, determine the intermediate assertion R" ;
   "prove the correctness of the associated lemmas (cf. Table I)" ;
   "add all new subproblems generated (cf. Table I) to
      problems-to-be-solved"
od
```

Depending on the program construct chosen for S, zero, one, or two new subproblems may be generated. As indicated by the above algorithm, the process continues until all subproblems have been solved.

The above discussion would appear to indicate that the only "invention" required in the process of deriving a program is in the choice of the appropriate construct for S. Unfortunately, this is not the case. Use of the composition rule (i.e., replacing S by $S_1;S_2$) requires the invention of the intermediate assertion R. Furthermore, it is usually the case that when S is to be replaced by an iteration, the input assertion P is unsuitable, either because it is not a loop invariant, or because it is not true that $\{P, \neg B \rightarrow Q\}$, or both. Thus, P must often be strengthened by means of the consequence axiom A7 or Q changed by means of the other consequence axiom A6. A more complete discussion of the problem of invention of loop assertions and the derivation of loop body code is given in Dijkstra [2], Gishen and Noonan [4].

The algorithm for carrying out a functional derivation can be stated as follows.

Algorithm D:

```
/* Given the specifications, organize them into a functional
   form where I represents the set of inputs, O represents
   the set of outputs, and f represents the mapping from the
   inputs to the outputs. */
functions-to-be-decomposed := [[ f ]]
while functions-to-be-decomposed ≠ empty do
   "using some strategy, choose a specific element of
      functions-to-be-decomposed, denoted f";
   "choose an appropriate decomposition for f";
   "prove the correctness of the decomposition
      (cf Table II)";
   "add all new functions generated to functions-to-be-decomposed;
od
```

Note that the functional model always generates a lemma to be proved and at least one new subproblem, except when the function can be achieved directly by an assignment (i.e., is already primitive). As indicated by the above algorithm, the process continues until all subproblems have been solved, that is, all functions have been fully elaborated. Like the axiomatic model "invention" is required in order to choose the appropriate decomposition for f.

It should be noted that Algorithms C and D are basically the same, except for terminology, arising from different statements of problem formulation, and for their lemmas and subproblem tables.

B. An Example: The Primes Problem

In the sections which follow, these algorithms are applied to the same problem, namely, the problem of finding and printing all primes less than or equal to a particular integer n. For both the axiomatic and functional models, the development of the appropriate specifications and the derivation of a correct algorithm are shown. The resulting programs are identical.

Informally, the program is to compute the set called primes = [[p1, p2, · · · , pm]] where each pi is a prime number less than or equal to n (≥2). More specifically, each pi is a positive integer greater than one and satisfies the condition that there does not exist an integer x, $1 < x < pi$, such that pi/x is an integer. Also, there are no prime numbers less than or equal to n that are not in the set primes.

```
1.        [$Z := A_{in} * B_{in}$  where $A_{in}$ and $B_{in} \geq 0$]
1.1       [$Z := 0$]
1.1.1      $Z := 0$;

1.2       [$Z := Z_{in} + A_{in} * B_{in}$  where $B_{in} \geq 0$]
1.2.1     while  $B > 0$  do
1.2.2        $Z := Z + A$;
1.2.3        $B := B - 1$
          od
```

Fig. 6. Functional correctness of simple multiplication.

```
1.   To prove
      [$Z := A_{in} * B_{in}$]  =  [$Z := 0$ ;  $Z := Z_{in} + A_{in} * B_{in}$]
                                =  [$Z := Z_{in} + A_{in} * B_{in}$]  ·  [$Z := 0$]
                                =  [$Z := 0 + A_{in} * B_{in} = A_{in} * B_{in}$]

2.   The proof that  [$Z := 0$]  is the function achieved by statement 1.1.1
     is obvious.

3.   To prove
      [$Z := Z_{in} + A_{in} * B_{in}$]  =  [while $B > 0$ do  $Z := Z + A$ ;
                                              $B := B - 1$  od]

     Case 1:  $B > 0$  :  must show
      [$Z := Z_{in} + A_{in} * B_{in}$]  =  [$Z := Z_{in} + A_{in}$ ;
                                             $B := B_{in} - 1$ ;
                                             $Z := Z + A * B$]

     Using a table, we get
      Step        Z                                    A        B
     Initially   $Z_{in}$                             $A_{in}$   $B_{in}$
       1.2.2     $Z_{in} + A_{in}$                    $A_{in}$   $B_{in}$
       1.2.3     $Z_{in} + A_{in}$                    $A_{in}$   $B_{in} - 1$
       1.2       $Z_{in} + A_{in} + A_{in} * (B_{in} - 1)$
                   $= Z_{in} + A_{in} * B_{in}$       $A_{in}$   $B_{in} - 1$

     Case 2:  $B \leq 0$  :  must show  [$Z := Z$ ]
      [$Z = Z_{in} + A_{in} * B_{in}$
         $= Z_{in} + A_{in} * 0$                  since $B \geq 0$ , $B \leq 0$
         $= Z_{in}$]
```

Fig. 7. Functional proof of simple multiplication.

In developing a solution, the following observations can be made. First, 2 is the first prime number and the only even prime. All other primes are odd and, therefore, it is necessary to test only odd numbers as additional prime number candidates. Second, any number that has a factor has a prime factor; therefore, it is necessary to divide the current candidate only by the primes already calculated. This means, however, that the primes must be saved as they are calculated.

In the specifications, primes will be treated as a set, a convenient choice for high level specification, but implemented as an array. The necessary properties that the implementation is valid could be stated and proved, but is irrelevant to the purposes of this paper.

C. Axiomatic Derivation of Primes

In the axiomatic model a program to be derived always starts out in the form

TABLE I
AXIOMATIC LEMMAS AND SUBPROBLEMS FOR PROBLEM $\{P\}\ S\ \{Q\}$

Construct Chosen for S	Lemma to be Proved	New Subproblems
1. $x := f$	$\{P \rightarrow Q\}$ where	
	P is obtained from	
	Q by replacing all	
	occurrences of x	
	by f	--
2. $S_1 ; S_2$	--	Determine R
		$\{P\}\ S_1\ \{R\}$
		$\{R\}\ S_2\ \{Q\}$
3. if B then S_1 else S_2 fi	--	$\{P, B\}\ S_1\ \{Q\}$
		$\{P, \neg B\}\ S_2\ \{Q\}$
4. while B do S_1 od	$\{P, \neg B \rightarrow Q\}$	
	loop terminates	$\{P, B\}\ S_1\ \{P\}$

TABLE II
FUNCTIONAL LEMMAS AND SUBPROBLEMS

Construct Chosen	Lemma	New Subproblems
1. Assignment	--	--
2. Sequence	$f \equiv [g; h]$	g, h
3. Choice (if p then g else h fi)	$f \equiv \begin{cases} g & \text{if } p \\ h & \text{if } \neg p \end{cases}$	g, h
4. Iteration (while p do g od)	$f \equiv \begin{cases} [g; f] & \text{if } p \\ \text{identity} & \text{if } \neg p \end{cases}$	g

$$\{P\}\ S\ \{Q\}$$

where P gives the required assumptions on the input and Q the intended function to be computed by S. Thus, the primes problem can be stated:

$$\{2 \leqslant n\}\ \ S\ \ \{\text{primes} = [[p \mid p \leqslant n, p \text{ is prime}]]\}. \tag{1}$$

In order to develop an axiomatic solution to this problem, a definition of the set of primes is needed. In order to distinguish program variables from function definitions, the latter will be underlined. Thus, the set of primes may be defined:

$$\begin{aligned} \underline{\text{primes}}\,(K) &= \underline{\text{empty}} \quad \underline{\text{if}}\ K < 2 \\ &= \underline{\text{primes}}\,(K-1) \cup [[K \mid \underline{\text{isprime}}\,(K)]] \\ &\qquad\qquad\qquad \underline{\text{otherwise}} \end{aligned} \tag{2}$$

$$\underline{\text{isprime}}\,(K) = (\forall p)\,(p \in \underline{\text{primes}}\,(K-1) \rightarrow K\ \underline{\text{rem}}\ p \neq 0). \tag{3}$$

Note that the operator rem returns the remainder of the value of the first expression divided by the second. Using these definitions, the primes problem can be restated as

$$\{n \geqslant 2\}\ \ S\ \ \{\text{primes} = \underline{\text{primes}}\,(n)\}. \tag{4}$$

The problem of "inventing" such definitions is beyond the scope of this paper. Much of the cleverness of the resulting program follows directly from the formulation of the definition of the problem.

The program can be refined by decomposing it into a sequence using Axiom A2 in order to compute the even and odd primes separately. Then using Axiom A1 on the first of the two statements, we arrive at the following program:

```
{n ≥ 2}
primes (1) := 2;  size := 1;
{primes = primes (2)}
S1
{primes = primes (n)}.                                   (5)
```

The program part S1 computes only odd primes; furthermore, S1 must contain a loop. Following Dijkstra [2] and Gishen and Noonan [4], the necessary loop invariant is developed. Note that for a given $y \geq 3$, if y is even then primes(y) = primes(y - 1). Since n may be either even or odd, the following loop invariant is used:

$$\{primes = primes\,(y - 1), odd\,(y), 3 \leq y \leq n + 2\}.$$

Thus, the loop (S1) can be further refined using Axioms A2, A1, and A3 to arrive at

```
{primes = primes (2)}
y := 3;
{primes = primes (y - 1), odd (y), 3 ≤ y ≤ n + 2}
while y ≤ n do
    S2
od
{primes = primes (n)}.                                   (6)
```

Note that proving the correctness of this elaboration requires the proof of the following lemmas.

1) Primes = primes (2) → (primes = primes (3 - 1), odd (3), $3 \leq n + 2$).

The proof is obvious since it is known that $n \geq 2$.

2) (Primes = primes (y - 1), odd(y), $3 \leq y \leq n+2$, $y > n$) → primes = primes (n).

Two cases arise. If n is even, then $y = n + 1$ and primes = primes ((n + 1) - 1) = primes (n). If n is odd, then

$$y = n + 2$$

and

$$\begin{aligned} primes &= primes\,((n + 2) - 1)\\ &= primes\,(n + 1)\\ &= primes\,(n) \end{aligned}$$

since n + 1 is even and, hence, not prime.

In a similar fashion, the code shown below is produced proceeding backward through S2:

```
{primes = primes (y - 1), odd (y), 3 ≤ y ≤ n}
 isprime := true; j := 2;
{primes = primes (y - 1), odd (y), 3 ≤ y ≤ n,
 isprime = (∀K) (1 ≤ K < j → y rem primes (K) ≠ 0)}
 while j ≤ SIZE and isprime do

    isprime := (y rem primes (j) ≠ 0);
    j := j + 1
  od
{primes = primes (y - 1), odd (y), 3 ≤ y ≤ n, isprime =
   isprime (y)}
 if isprime then
   size := size + 1 ;
   primes (size) := y
 fi ;
{primes = primes (y + 1) = primes (y), odd (y), 3 ≤ y ≤ n}
 y := y + 2
{primes = primes (y - 1), odd (y), 3 ≤ y ≤ n + 2}         (7)
```

The final program with its intermediate assertions as documentation is shown in Fig. 8.

D. Functional Derivation of Primes

In the functional approach, the problem must be specified as a function from a set of inputs to a set of outputs. As before, this can be stated

$$primes := [[p \mid p \leq n, p \text{ is prime}]]. \qquad (8)$$

As with the previous solution, the even and odd primes are computed separately and the prime number candidates are divided only by prime numbers. Under these conditions, the functional specifications may be rewritten as

$$primes := [[2]] \cup oddprimes\,(3, n) \qquad (9)$$

using the functions:

$$\begin{aligned} oddprimes\,(l, u) &= [[y \mid y = l \wedge isprime\,(y)]] \cup oddprimes\,(l + 2, u) && \text{if } l \leq u\\ &= empty && \text{if } l > u \end{aligned}$$

$$isprime\,(x) = (\forall p)\,(p \in oddprimes\,(3, x - 1) \rightarrow x \text{ rem } p \neq 0).$$

The functional specification (9) can be decomposed into two functions, the first of which initializes the basic data and the second defines the iteration that does the bulk of the calculation.

```
1.    [primes := [[2]] ∪ oddprimes (3, n_in)]
1.1   primes (1) := 2 ; size := 1 ;
1.2   y := 3 ;
1.3   [primes := primes_in ∪ oddprimes (y_in, n_in)].    (10)
```

The implicit loop in computing odd primes can now be made explicit:

```
1.3     [primes := primes_in ∪ oddprimes (y_in, n_in)]
1.3.1   while  y ≤ n do
1.3.2     [primes := primes_in ∪ [[y_in | isprime (y_in)]] ;
           y := y_in + 2]
        od.                                               (11)
```

As with all expansions, the associated lemmas given in Table II must be proven. Since this refinement is not obvious, the expansion is verified. Since the refinement is a loop, three lemmas must be proved.

1) Does the loop terminate? Yes, since y is incremented by 2 for each iteration and is bounded above.

2) Whenever the loop test is true ($y \leq n$), is the loop body composed with the intended function of the loop equivalent to the intended function of the loop? This is demonstrated using a trace table.

```
{ n ≥ 2 }
 primes (1) := 2 ;  size := 1 ;
{primes  =  primes (2)}
 y  :=  3 ;
{primes = primes (y-1), odd (y), 3 ≤ y ≤ n + 2}
 while  y ≤ n  do
     isprime := true ;  j := 2 ;
     {primes = primes (y-1), odd (y), 3 ≤ y ≤ n ,
          isprime = (∀K) (1 ≤ K < j → y rem primes (K) ≠ 0)}
     while j ≤ size and isprime do
          isprime := (y rem primes (j) ≠ 0) ;
          j := j + 1
     od ;
     {primes = primes (y - 1), odd (y), 3 ≤ y ≤ n,
          isprime = isprime (y)}
     if  isprime  then
          size :=  size + 1 ;
          primes (size) :=  y
     fi ;
     {primes = primes (y) = primes (y + 1), odd (y), 3 ≤ y ≤ n}
          y  :=  y + 2
     {primes = primes (y - 1), odd (y), 3 ≤ y ≤ n + 2}
od
{primes  =  primes (n)}
```

Fig. 8. Axiomatic solution of primes problem.

Step	Primes	y
initially	$primes_{in}$	y_{in}
loop body	$primes_{in} \cup [[y_{in} \mid \underline{isprime}(y_{in})]]$	$y_{in} + 2$
loop function	$primes_{in} \cup [[y_{in} \mid \underline{isprime}(y_{in})]]$	
	$\cup\ \underline{oddprimes}(y_{in} + 2, n_{in})$	–
	$= primes_{in} \cup \underline{oddprimes}(y_{in}, n_{in})$.	

Since the final value for primes is the same as the intended function of (11), this case is proved.

3) Whenever $y > n$, is the intended function of the loop an identity? Yes, since $y > n$, the set $\underline{oddprimes}(y_{in}, n)$ is empty. Thus,

$$primes = primes_{in},$$

and this case is proved.

Although the correctness of each successive expansion is not verified, it should be clear that it is both possible and often helpful to do so.

The solution process is continued by expanding the loop body given in (11).

```
1.3.2    [primes := $primes_{in} \cup [[y_{in} | \underline{isprime}(y_{in})]]$;
            y := $y_{in}$ + 2]
1.3.2.1  [isprime := $\underline{isprime}(y_{in})$]
1.3.2.2  if isprime then
1.3.2.3     size := size + 1 ;
1.3.2.4     primes (size) := y
         fi ;
1.3.2.5  y := y + 2.                                          (12)
```

```
1.           [primes := [[2]] ∪ oddprimes (3, n)]
1.1          primes (1) := 2 ;  size := 1 ;
1.2          y := 3 ;
1.3          [primes := primes_in ∪ oddprimes (y_in, n_in)]
1.3.1        while  y ≤ n  do
1.3.2            [primes := primes_in ∪ [[y_in | isprime (y_in)]] ;
                  y := y_in + 2]
1.3.2.1          [isprime := isprime (y_in)]
1.3.2.1.1        isprime := true ;  j := 2 ;
1.3.2.1.2        while  j  ≤  size and isprime  do
1.3.2.1.3            isprime := (y rem primes (j) ≠ 0) ;

1.3.2.1.4            j := j + 1
                 od ;
1.3.2.2          if  isprime  then
1.3.2.3              size := size + 1 ;
1.3.2.4              primes (size) := y
                 fi ;
1.3.2.5          y := y + 2
             od
```

Fig. 9. Functional solution of primes problem.

The final expansion is the loop necessary to calculate isprime.

```
1.3.2.1      [isprime := $\underline{isprime}(y_{in})$]
1.3.2.1.1    isprime : = true ; j := 2 ;
1.3.2.1.2    while j ≤ size and isprime do
1.3.2.1.3       isprime := (y rem primes (j) ≠ 0) ;
1.3.2.1.4       j := j + 1
                od.                                        (13)
```

The complete program with its intermediate functions as documentation is given in Fig. 9. It should be noted that this program is identical to the one given in Fig. 8.

IV. Comparison Between the Two Models

In what follows, some of the similarities and differences between the two models and their associated correctness and derivation approaches are discussed.

A. Similarities

Formal Models of Individual Program Constructs: Both approaches are based upon formal tractable mathematical models for specific sets of program constructs in isolation (not as operational models of the interrelationships of program constructs at run time). The models for the individual constructs give an indication of the complexity of the semantics of the constructs and, thus, yield a good motivation for the choice of a set of programming language constructs for use in writing provably correct programs. They both deal with partial correctness only; proof of termination is a separate issue and identical techniques can be used in both models.

Stepwise Derivation and Correctness: Rules for derivation and correctness are based on the application of the particular constructs as they are decomposed in the development process and composed in the abstraction process. Both techniques are applicable in a stepwise manner, at various levels in the correctness and development process, dealing with only small segments of code, and expanding subspecifications in a step-by-step manner. In this way, they also make excellent documentation techniques, each subspecification being useful as a high level comment about the code expanded below it.

Invention: As methodologies for proving correctness, both approaches require some invention in the creation of the loop invariant and the intended loop function, respectively. If these are not given, there is no practical way of generating them which is guaranteed to succeed in a reasonable amount of time. A great deal of work has been done on heuristics for finding loop invariants (Wegbreit [12]). Some results have recently been published in generating intended loop functions (Blikle [1]).

B. Differences

Underlying Mathematics: The underlying mathematics of each of the models is different. The axiomatic approach uses the predicate calculus while the functional approach uses the concepts of function composition and equivalence. Consider the rules for correctness given in Section II. One set of rules uses logical consequence and the logical operators of the predicate calculus, while the other uses function composition (decomposition for derivation) and function equivalence.

Statement of the Specification: The functional approach states the specifications and subspecifications as functions from the input value space to the output value space. It is a mathematical function in the strict sense. The axiomatic approach organizes the specifications and subspecifications into Boolean functions represented by assertions on program variables where the input assertion is a set of status relations among the input program variables and the output assertion yields true or false depending upon whether the output variables satisfy a specific relationship with the input variables. In illustration, consider the followng simple program:

```
I := 1 ;
I := I + 1.
```

The format for the axiomatic and functional approaches are given below:

```
{true}       1.      [I := 2]
 I := 1      1.1     [I := 1]
{I = 1}      1.1.1    I := 1
 I := I + 1  1.2     [I := I_in + 1]
{I = 2}      1.2.1    I := I + 1.
```

In the axiomatic approach, each assertion shows what is true about the state of the variables at the particular point in the program where the assertion appears. The assertion is given in terms of a relationship between the variables involved, e.g., $\{I = 1\}$. In the functional approach, the function defines the effect a particular set of statements has with respect to its set of input and output values. Thus, the statement $I := I + 1$ is defined by the function $[I := I_{in} + 1]$ (shorthand for the normal function notation $(I, I_{in} + 1) \in f$) where I_{in} represents the first element of the tuple and I represents the second element. Thus, the functional specification is not in terms of the variables at all but the values of the variables, i.e., I_{in} and I represent different values of the same program variable at point of input and output of exit to the program segment specified. The function gives the relationship between the two value sets.

In contrast, an assertion is a mapping from the current values of the variables into [[true, false]]. Thus, the final output assertion must somehow capture both the original input values as well as the final output values in a single assertion over the current values of the program variables. Two approaches can be taken in the event that the program destroys the values of the original input variables. One approach, taken in Fig. 3, is to introduce artificial variables into the program to hold those input values which are to be destroyed. The other approach is to add free variables to the assertions and add some conventions or rules for binding these variables to their appropriate values. This latter approach would have to be used for the program in Fig. 6. Both of these are unnecessary in the functional model. See Fig. 10 for the specifications for both programs. Note that the functional specifications for the two programs are the same. This is because only the final value of z is of interest.

In addition, the functional model is variable-free in that a function abstracts from the specific program variables used. In this sense, the assignment statements $X := X + 1$ and $I := I + 1$ represent the same function, namely, $f(Z) = Z + 1$. In the axiomatic model the assertions refer specifically to actual program variables. Although a specific output assertion could be parameterized (in a manner analogous to the axioms developed for procedures (Hoare [7]) so as to permit the substitution of actual program variables for symbolic parameters, such an approach would considerably complicate proofs in the axiomatic model.

Scope of Specification: A functional specification defines the state of affairs of only the program part for which it is the intended function. For example, the function $[I := I_{in} + 1]$ describes only the behavior of the statement $I := I + 1$. However, in the axiomatic model, the assertion $\{I = 2\}$ depends not only on the statement $I := I + 1$, but also on the previous history of I.

Any change in a program not affecting a particular program segment implies that the functional specification for the segment need not be changed and no new proof of functional correctness is required for that segment. In addition, a different implementation of a functional specification can be substituted without changing any proofs of correctness in the remainder of the program. For example, as shown in Fig. 10, the two multiplication programs have the same functional specification, and one program could be substituted for the other in a larger program without reproving the larger program.

This is not true in practice for the axiomatic model. An assertion about a variable usually depends on the history of the use of that variable and on its interdependence on other variables. Consider the axiomatic specifications for the two multiplication programs. The specification for Fig. 6 is valid for Fig. 3, but not vice versa. Because the value of the variable B is set to zero in the program in Fig. 6, the output specifica-

	Functional	Axiomatic	
		P	Q
Fig. 3	[[(A,B) , A*B|A,B $\geqslant$ 0]]	A,B $\geqslant$ 0	Z = A*B
Fig. 6	[[(A,B) , A*B|A,B $\geqslant$ 0]]	A,B $\geqslant$ 0,B_0=B	Z = A*B_0

Fig. 10. Specifications for the multiplication programs.

tion Z = A * B implies Z = 0 which is incorrect. For an output assertion to be stable under program modification it requires that the values of all input variables be bound to free variables in the output assertion. Such an approach would lead to significantly longer assertions and, thus, greatly complicate the proofs of the resulting verification conditions.

Another way that the assertions in an axiomatic specification are nonlocal is that they normally contain global information about nonlocally affected variables. For example, if the multiplication program was embedded in an exponentiation program, all the assertions associated with the multiplication portion of the program might be embedded in other relations concealing the true function of the multiplication.

Bottom-Up Correctness: An added effect of this difference is that given a program without any functional specifications, the intended function for any statement can be defined depending only on the specific subhierarchy of the program, i.e., functional correctness can be approached bottom-up. Suppose the functional correctness of the program in Fig. 11 is to be proved bottom-up. The functional equivalent of the loop body is

$$[\, I := I_{in} + 1 \,]$$

and the proof is trivial. The functional equivalent of the loop is

$$[\, I := \max(I_{in}, N_{in}) \,]$$

or

$$\begin{aligned}[\, I &:= N_{in} && \text{if } I_{in} < N_{in} \\ &:= I_{in} && \text{if } I_{in} \geqslant N_{in} \,].\end{aligned}$$

The proof that the loop is equivalent to the above function requires proving that the loop body is equivalent to its function.

Now consider the case for the axiomatic method; given

$$\{P\} \;\; I := I + 1 \;\; \{Q\}$$

P and Q must be found. Given Q, P can be found from Q via the assignment axiom A1. Unfortunately, there is no way to determine Q; in a sense Q must contain a great deal of historical information about the use of the variable I as well as its relationship to the nonlocally referenced N. Suppose, however, the straightforward approach adopted for the functional method was tried:

$$\{\underline{\text{true}}\} \;\; I := I + 1 \;\; \{I = I_{in} + 1\}.$$

However, given the invariant for the loop $\{I \leqslant N\}$, a proof of correctness requires showing

$$\{P, B\} \;\; I := I + 1 \;\; \{P\}$$

or

$$\{I \leqslant N, I < N\} \;\; I := I + 1 \;\; \{I \leqslant N\}.$$

```
{N ≥ 0}

 I := 0

{I ≤ N}

while  I < N  do

        I := I + 1

od

{I = N}
```

Fig. 11. Program to compute I = N.

Thus, without considering the loop as a whole, there is no practical way of determining the correct loop body postcondition; that is, there is no practical way to assure that the choice of the postcondition is sufficiently strong to be of value in the proof of correctness of the loop body.

In the functional approach, any such bottom-up process is guaranteed to be relevant to the larger construct. In fact, in the program in Fig. 11, given the program as a whole, the intended function of the loop is actually

$$[\, I := N_{in} \quad \text{if } I_{in} \leqslant N_{in} \,].$$

Note that this is weaker than the one found in the bottom-up process. This is necessarily so, since the top-down process can consider only the relevant input domain ($N \geqslant 0$) instead of the entire input domain (N an integer).

Proof of correctness by subgoal induction (Morris [11]) provides a method of proving bottom-up correctness for loops in the framework of the axiomatic model. This is actually achieved by using a technique based on functional correctness which dispenses with the need to find the loop invariant and uses the intended function instead.

However, it should be noted that even in the functional model top-down proofs are easier. Because the intended function is more specific than the functional equivalent of the program, the algebraic manipulations are greatly simplified in proving the necessary functional equivalences. Consider the multiplication programs defined earlier. If the development were done bottom-up, the function computed for Fig. 3 would be

$$[[(A,B,Y,Z),\; (A,B,O,A*B)) \;\; A_{in}, B_{in} > 0]]$$

since it would not be known whether A, B, and Y are relevant outputs or required for input to another part of the program. This can become pretty complicated; for example, consider the problem of finding the intended function of a binary search program bottom-up if it is not known that the input array is sorted.

C. Interrelationship

There is an interesting connection between the two models. The intended function of a loop may be easily converted to a loop invariant; namely, $f(x) = f(x_{in})$ is the loop invariant (Mills [9], Misra [10]). In the multiplication example, the functional specification for the loop is $Z := Z_{in} + A_{in} * B_{in}$; thus, the loop invariant is

$$f(x) = f(x_{in})$$
$$Z + A * B = Z_{in} + A_{in} * B_{in}$$
$$Z = Z_{in} + A_{in} * B_{in} - A * B$$
$$Z = Z_{in} + A * (B_{in} - B), \quad \text{since } A = A_{in}$$
$$Z = A * (B_{in} - B),$$

since $Z_{in} = 0$ from the initial assertion and the composition of the statements preceding the loop. In the program in Fig. 3, the variable B plays the role of B_{in} and the variable Y plays the role of B.

It should be noted that

$$Z = Z_{in} + A * (B_{in} - B)$$

is actually the loop invariant if one were to treat the loop independently of the prior statements in the program. That is, in general, the loop assertion is simplified because one normally takes advantage of the variable environment in which the current statement exists. The translation from the intended function to the loop invariant always yields the most general form of the loop invariant. The development of the loop invariant from the intended function has been studied independently and quite thoroughly (Misra [10]).

Given the inductive assertion, it has been shown by Morris [11] that a relational form of the intended function can be constructed. This demonstrates the theoretical duality of the two approaches. However, when constructing the intended function from the loop invariant, two problems result. First, the form of the function is a relation rather than a mapping from inputs to outputs. The second problem is centered around the fact that the normal loop invariant contains a history of the prior usage of its variables. Thus, the intended function derived from it reflects this history and does not represent the intended function of the associated statement independent of the rest of the program. For example, unless the loop invariant

$$Z = Z_{in} + A * (B_{in} - B)$$

rather than

$$Z = A * (B_{in} - B)$$

is used, the intended function

$$Z := Z_{in} + (A_{in} * B_{in})$$

cannot be derived. Instead,

$$Z = A_{in} * B_{in}$$

which assumes $Z_{in} = 0$ would result.

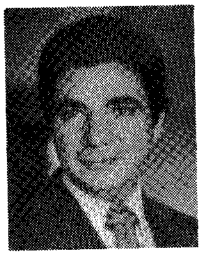
V. Conclusion

It should be clear from the previous discussion that both models yield a methodology of program derivation and correctness. The approaches have a great deal in common, but they are different; the axiomatic approach emphasizes the relations between the variables, and the functional approach emphasizes the independent variable-free functions performed by the various program subpieces.

Is it not clear which approach is more effective in an operational environment. It may, in fact, depend upon the particular environment and the problems that arsie. Enough is still not known about the kinds of errors designers and programmers make in different environments. Certainly, one could be used in formally deriving the program and the other could be used as a commenting aid; e.g., use the functional approach in the development to aid in the partitioning and modularization of the independent program parts and use assertions as comments to aid the programmer in understanding the relationships between the variables. In either case, the models appear to complement each other in the insight they provide to the developer.

References

[1] A. Blikle, "An analytical approach to the verification of iterative programs," *Inform. Processing* 77, pp. 285-290, 1977.
[2] E. W. Dijkstra, *A Discipline of Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1976.
[3] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Appl. Math.*, vol. 19, 1967, pp. 19-32.
[4] J. S. Gishen and R. E. Noonan, "Toward a methodology for the formal derivation of programs," *IEEE Trans. Software Eng.*, to be published.
[5] S. L. Hantler and J. C. King, "An introduction to proving the correctness of programs," *Comput. Surveys*, vol. 8, pp. 331-353, Sept. 1976.
[6] C. A. R. Hoare, "An axiomatic basis for computer programming," *CACM*, vol. 12, pp. 576-583, Oct. 1969.
[7] —, *Procedures and Parameters, An Axiomatic Approach in Symposium on the Semantics of Algorithmic Languages*, E. Engler, Ed. Berlin, Germany: Springer, 1971, pp. 102-116.
[8] H. D. Mills, "Mathematical foundations for structured programming," IBM Fed. Syst. Div., FSC 72-6012, 1972.
[9] —, "The new math of computer programming," *CACM*, vol. 18, Jan. 1975.
[10] J. Misra, "Some aspects of the verification of loop computations," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 478-486, Nov. 1978.
[11] J. H. Morris and B. Wegbreit, "Subgoal induction," *CACM*, vol. 20, pp. 209-222, Apr. 1977.
[12] B. Wegbreit, "The synthesis of loop predicates," *CACM*, vol. 17, pp. 102-112, Feb. 1974.



Victor R. Basili was born in Brooklyn, NY. He received the B.S. degree in mathematics from Fordham College, Bronx, NY, the M.S. degree in mathematics from Syracuse University, Syracuse, NY, and the Ph.D. degree in computer science from the University of Texas, Austin, in 1961, 1963, and 1970, respectively.

From 1963 to 1967 he was a member of the faculty of the Department of Mathematics and Computer Science, Providence College, Providence, RI. Since 1970 he has been a member of the faculty of the Department of Computer Science, University of Maryland, College Park, where he is presently an Associate Professor. He has been involved in the design and development of several software projects, including the SIMPL family of structured programming languages, the graph algorithmic language GRAAL, and the SL/1 language for the CDC STAR. He is currently involved in the measurement and evaluation of software development at NASA/Goddard Space Flight Center. He has acted as a consultant for several industrial organizations and government agencies, including IBM, GE, CSC, NRL, NSWC, and NASA.

Dr. Basili is a member of the Association for Computing Machinery, the IEEE Computer Society, and the American Association of University Professors.

Robert E. Noonan was born in Rahway, NJ, on June 4, 1944. He received the A.B. degree summa cum laude in mathematics from Providence College, Providence, RI, and the M.S. and Ph.D. degrees in computer science from Purdue University, Lafayette, IN, in 1966, 1968, and 1971, respectively.

In 1971 he joined the Department of Computer Science, University of Maryland, College Park. Since 1976 he has been an Associate Professor of computer science with the College of William and Mary, Williamsburg, VA. He has served as a consultant for the U.S. Naval Weapons Laboratory, Dahlgren, VA, the Maryland State Department of Education, and James City County, VA. His research interests include programming languages and compilers, programming methodology, and software engineering. He has been involved in the design and development of the MISTRO compiler generation system and of the CGGL (compiler code generation) language.

Dr. Noonan is a member of the IEEE Computer Society and the Association for Computing Machinery.