AD-A053 629 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
A COMPARISON OF THE AXIOMATIC AND FUNCTIONAL MODELS OF STRUCTUR--ETC(U)
FEB 78 V R BASILI, R E NOONAN AFOSR-77-3181

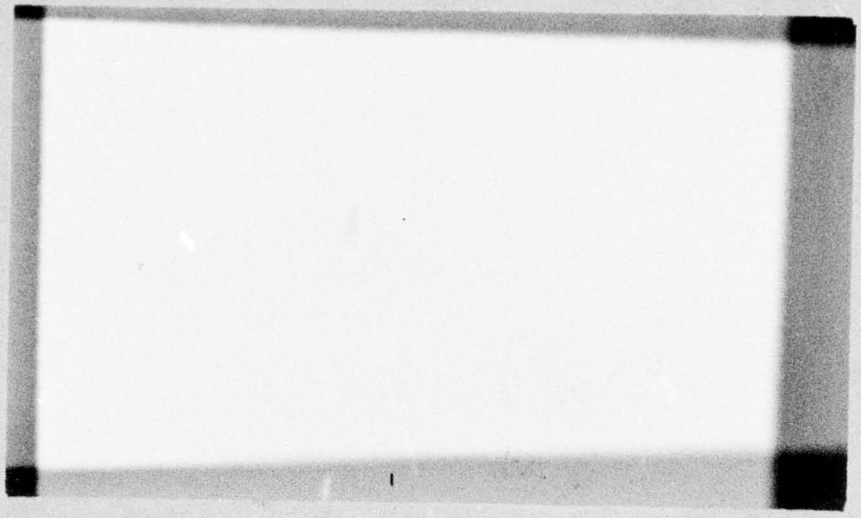
UNCLASSIFIED TR-630 AFOSR-TR-78-0737 NL

1 OF 1
AD A053629

END
DATE FILMED
6-78
DDC

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742



AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

Technical Report TR-630 February 1978

A COMPARISON OF THE AXIOMATIC AND FUNCTIONAL MODELS OF STRUCTURED PROGRAMMING*

Victor R. Basili[1] and Robert E. Noonan[2]

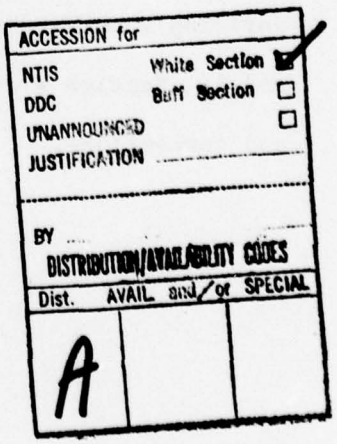
[1]Department of Computer Science, University of Maryland, College Park, Maryland

[2]Department of Mathematics and Computer Science, College of William and Mary, Williamsburg, Virginia

*This research was supported in part by the Air Force Office of Scientific Research, Grant AFOSR77-31 81, to the University of Maryland.

ABSTRACT

This paper discusses the 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.



1. INTRODUCTION

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 there is a rule for verifying the correctness of the expansion from its specification or intended function. This paper discusses two such models, the axiomatic model [Floyd, 1967; Hoare, 1969] and the functional model [Mills, 1972, 1975]. 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 3 gives the algorithm for deriving a correct program and contains an example derivation using each model. Section 4 compares the two models: their similarities, differences and connections.

2. BASIC MODELS OF PROGRAM CONSTRUCTS AND CORRECTNESS

In this section, the two models are defined and the semantic rules given for each of the following program constructs 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. In order to construct this hierarchy, we define a prime program as a structured program with no structured subprograms consisting of more than one statement. The following algorithm is used to construct this hierarchy:

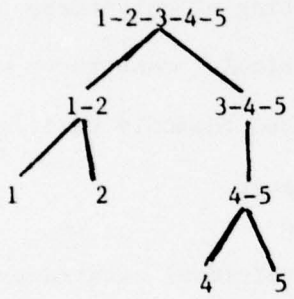
while the program consists of more than a single logical statement do
    "find a prime subprogram in the current program;"
    "reduce the prime program to a single logically equivalent

statement - this reduction is essentially a syntatic grouping, i.e., a compound statement (note that the resulting program consists of fewer statements)"

od

This algorithm produces a hierarchy of prime programs in which each iteration of the loop produces a higher level in the hierarchy.

As an example, consider the program given in Figure 2.1. The hierarchy of prime programs is the following, where statements are indicated by their line number:



2.1 Axiomatic Correctness

The particular model of axiomatic correctness that we use here is due to [Hoare, 1969].

> 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 pre-conditions of successful use can be specified by the same type of general assertion as is used to describe the results on termination.

FIGURE 2.1

SIMPLE MULTIPLICATION PROGRAM

```
1.   Z  :=  0 ;

2.   Y  :=  B ;

3.   while  Y  >  0    do

4.          Y :=  Y - 1 ;

5.          Z :=  Z + A

     od
```

The notation

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

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, 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. In this paper, we ignore the problem of termination since the techniques used are identical for both models.

To prove that S is correct with respect to P, Q, the first order predicate logic is used together with axioms for each program construct. These latter axioms and rules of inference are given in Figure 2.2. A complete explanation of these axioms and rules of inference can be found in [Hoare, 1969]. In the iteration axiom 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 post assertion and the hierarchy of prime programs has been produced. Since each prime program in the hierarchy has both a pre and post assertion, the correctness of each piece is proved using the rules given in Figure 2.2 based on the form of the prime program. This demonstrates the partial correctness of the program.

In practice, however, every statement in a program is rarely tagged with both its pre and post-conditions. The minimum effective requirement is

FIGURE 2.2

HOARE'S RULES OF INFERENCE

Assignment

A1 $\{P(f)\}\ x := f\ \{P(x)\}$

where P(f) is obtained from P(x) by substituting f for all occurrences of x.

Composition

A2 If $\{P\}$ S1 $\{R\}$ and $\{R\}$ S2 $\{Q\}$,

then $\{P\}$ S1; S2 $\{Q\}$.

Iteration

A3 If $\{P \wedge B\}$ S $\{P\}$,

then $\{P\}$ while B do S od $\{P \wedge \neg B\}$

Condition

A4 If $\{P \wedge B\}$ S1 $\{Q\}$ and $\{P \wedge \neg B\}$ S2 $\{Q\}$,

then $\{P\}$ if B then S1 else S2 fi $\{Q\}$

A5 If $\{P \wedge B\}$ S1 $\{Q\}$ and $P \wedge \neg B \rightarrow Q$,

then $\{P\}$ if B then S1 fi $\{Q\}$

Consequence

A6 If $\{P\}$ S $\{Q\}$ and $Q \rightarrow R$, then $\{P\}$ S $\{R\}$.

A7 If $\{Q\}$ S $\{R\}$ and $P \rightarrow Q$, then $\{P\}$ S $\{R\}$.

to be given the pre and post-conditions for the entire program and the loop invariant for each loop. In this case, the pre and post conditions are generated by a top-down process.

To produce the assertions, first produce the hierarchy of prime programs. 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 pre-condition of each statement is generated by a top-down, recursive algorithm from the post condition (note that the output assertion of the entire program is assumed to be given):

1. Assignment: x:= f. If the post-condition is P(x), then the pre-condition is P(f) as given in rule A1 of Figure 2.2.

2. Sequence: S1 ; S2. If the post-condition is Q, then the process is involved recursively to find the pre-condition of S2, namely, R. This process is then repeated on S1 given R as its post-condition.

3. Iteration: while B do S od. Note that it is assumed that the loop invariant P is given. If Q is the post condition, it must be proved (as part of the proof process) that P and $\neg$ B $\rightarrow$ Q. P is the pre-condition of the loop. {P and B} is the given pre-condition of S and P the post-condition of S. The process of finding pre-conditions is continued on S.

4. Choice: if B then S1 else S2 fi. If Q is the post-condition of the if, then the process is repeated on S1 and S2

with Q as their post-conditions, yielding P1 and P2 as their respective pre-conditions. The pre-condition of the *if* is $(B \rightarrow P1) \wedge (\neg B \rightarrow P2)$. Note that this condition can normally be greatly simplified. For example, if P1 is of the form $P \wedge B$ and P2 is of the form $P \wedge \neg B$ then the pre-condition is P.

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

In order to illustrate this method, the correctness of the program given in Figure 2.3 is proven. This program computes the product of two natural numbers A, 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. The proof of the program given in Figure 2.4 uses line numbers to refer to program parts of more than one statement.

2.2 Functional Correctness

The model for functional correctness was developed by [Mills, 1972, 1975]. Here, the intended function of a program is stated as a functional

FIGURE 2.3

AXIOMATIC CORRECTNESS OF SIMPLE MULTIPLICATION

```
{ 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 }
```

FIGURE 2.4

AXIOMATIC PROOF OF SIMPLE MULTIPLICATION

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

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 a vector of inputs I and a vector of outputs 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 substitution of values 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 Figure 2.5. Note that the rule for assignment given in Fl 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 spaces are always distinct. 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

FIGURE 2.5

FUNCTIONS COMPUTED BY PROGRAM CONSTRUCTS

Assignment

F1 $\quad y = f(x)$

Composition

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

Iteration

F3 $\quad f(x) = [\underline{\text{while}}\ p\ \underline{\text{do}}\ g\ \underline{\text{od}}](x)$

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

Conditions

F4 $\quad f(x) = [\underline{\text{if}}\ p\ \underline{\text{then}}\ g\ \underline{\text{else}}\ h\ \underline{\text{fi}}](x)$

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

F5 $\quad f(x) = [\underline{\text{if}}\ p\ \underline{\text{then}}\ g\ \underline{\text{fi}}](x)$

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

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 prime program in the hierarchy has an intended function associated with it. Then merely show the equivalence of each prime program to its intended function using the rules given in Figure 2.5. This yields the proof of partial correctness of the program.

In practice, however, every prime program is rarely tagged with 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 the given intended functions with their associated subprograms consisting of the top level prime program and its subhierarchy down to the level of any given intended functions. In this process the functions computed by the subhierarchy may be effectively created using a top-down recursive trace algorithm through the levels of the subhierarchy as follows:

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 <u>then</u> part and the <u>else</u> part, and continue the trace process through each part separately.

FIGURE 2.6

FUNCTIONAL CORRECTNESS OF SIMPLE MULTIPLICATION

[Z := Ain * Bin , where Ain , Bin $\geq$ 0]

[Z := 0]

1. Z := 0;

[Z := Zin + Ain * Bin, where Bin $\geq$ 0]

2. **while** B > 0 **do**

3. Z := Z + A;

4. B := B - 1

od

FIGURE 2.7

FUNCTIONAL PROOF OF SIMPLE MULTIPLICATION

1. To prove

$$\lceil Z := Ain * Bin \rceil = \lceil Z := 0 ;\ Z := Zin + Ain * Bin \rceil$$
$$= \lceil Z := Zin + Ain * Bin \rceil \cdot \lceil Z := 0 \rceil$$
$$= \lceil Z := 0 + Ain * Bin = Ain * Bin \rceil$$

2. The proof that $\lceil Z := 0 \rceil$ is the function achieved by statement 1 is obvious.

3. To prove

$$\lceil Z := Zin + Ain * Bin \rceil = \lceil \underline{while}\ B > 0\ \underline{do}\ Z := Z + A ;$$
$$B := B - 1\ \ \underline{od} \rceil$$

Case 1: $B > 0$: must show

$$\lceil Z := Zin + Ain * Bin \rceil = \lceil Z := Zin + Ain ;$$
$$B := Bin - 1 ;$$
$$Z := Z + A * B \rceil$$

Using a table, we get

Step	Z	A	B
Initially	Zin	Ain	Bin
3	Zin + Ain	Ain	Bin
4	Zin + Ain	Ain	Bin - 1
f	Zin + Ain + Ain * (Bin - 1) = Zin + Ain * Bin	Ain	Bin - 1

Case 2: $B \leq 0$: must show / Z := Zin /

$$\lceil Z = Zin + Ain * Bin$$
$$= Zin + Ain * 0 \qquad \text{since } B \geq 0 ,\ B \leq 0$$
$$= Zin \rceil$$

In order to illustrate this method, the partial correctness of the program given in Figure 2.6 is demonstrated in Figure 2.7. Note that this program is nearly identical to the one given in Figure 2.3.

3. 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.

3.1 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 generating possibly both 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.

The algorithm for carrying out an axiomatic derivation can be stated as follows (note sets are denoted as [[...]]):

```
/*  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.      */

P := "assertion of input specifications"
Q := "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 problem
```

TABLE 3.1

AXIOMATIC LEMMAS AND SUBPROBLEMS

FOR PROBLEM $\{P\}\ S\ \{Q\}$

Construct Chosen	Lemma to be Proved	New Subproblems
1. $x := f$	$P \rightarrow Q$ where Q 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. <u>if</u> B <u>then</u> S_1 <u>else</u> S_2 <u>fi</u>	--	$\{P \wedge B\}\ S_1\ \{Q\}$ $\{P \wedge \neg B\}\ S_2\ \{Q\}$
4. <u>while</u> B <u>do</u> S_1 <u>od</u>	$P \wedge \neg B \rightarrow Q$ loop terminates	$\{P \wedge B\}\ S_1\ \{P\}$

to be solved, denoted {P} S {Q}" ;

"choose an appropriate program construct for S" ;

"prove any associated lemma (cf. table 4.1)" ;

"add any new subproblems generated (cf. table 4.1)
to problems-to-be-solved"

od

Depending on the program structure 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_1S_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 $P \wedge \neg B \rightarrow Q$ or both. Thus, P must often be strengthened by means of the consequence axiom, A7, or Q changed by means of axiom A6. A more complete discussion of the problem of invention of loop assertions and the derivation of loop bodycode is given in [Dijkstra, 1976; Gishen and Noonan, 1978].

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

/* Given the specifications, organize them into a functional form where I represents the set of inputs, O represents

TABLE 3.2

FUNCTIONAL LEMMAS AND SUBPROBLEMS

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

```
        the set of outputs and  f  represents the mapping from the
        inputs to the outputs.   */

    I,  O,  f  := convert-to-function-format (specifications)
    functions-to-be-decomposed := [[f]]

    while     functions-to-be-decomposed ≠ empty do
        "choose a function from the set and decompose it into one
          of the acceptable program constructs" ;
        "prove the decomposition is correct (cf. Figure 4.2)" ;
        "add any nonprimitive functions 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.

3.2 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.

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 ($\geq$ 2).

More specifically, each pi is a positive integer and satisfies the condition that there does not exist an integer x, $1 < x < p$, 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.

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.

3.3 Axiomatic Derivation of Primes

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

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

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

(3.3.1) $\{2 \leq n\} \quad S \quad \{primes = [[p \mid p \leq n, p \text{ is prime}]]\}$.

In order to develop an axiomatic solution to this problem, 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:

(3.3.2) $\underline{primes}\ (K) = \underline{empty}$ $\underline{if}\ K < 2$

$= \underline{primes}\ (K-1) \cup [[K \mid \underline{isprime}\ (K)]]$ $\underline{otherwise}$

(3.3.3) $\underline{isprime}\ (K) = (\forall p)\ (p \in \underline{primes}\ (K-1) \cdot K\ \underline{rem}\ p \neq 0)$

Using these definitions, the primes problem can be restated as:

(3.3.4) $\{n \geq 2\}$ S $\{primes = \underline{primes}\ (n)\}$.

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:

(3.3.5) $\{n \geq 2\}$

primes (1) := 2; size := 1;

$\{primes = \underline{primes}\ (2)\}$

S1

$\{primes = \underline{primes}\ (n)\}$

The program part S1 computes only odd primes; furthermore, S1 must contain a loop. Following [Dijkstra, 1976] and [Gishen and Noonan, 1978], the necessary loop invariant is developed. Note that for a given $y \geq 3$, if y is even then $\underline{primes}\ (y) = \underline{primes}\ (y-1)$. Since n may be either even or odd, the following loop invariant is used:

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

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

(3.3.6) $\{primes = \underline{primes}\ (2)\}$

y := 3;

{primes = primes (y-1), odd (y), y ≤ n + 2}

while y ≤ n do

S2

od

{primes = primes (n)}

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 ≤ n + 2)

   The proof is obvious since it is known that n ≥ 2.

2. (primes = primes (y-1), odd (y), y ≤ 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

   primes = primes ((n + 2) - 1)
   = primes (n + 1)
   = primes (n) since n + 1 is even and hence, not prime.

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

(3.3.7) {primes = primes (y-1), odd (y), y ≤ n}

isprime := true ; j := 2 ;

{primes = primes (y-1), odd (y), 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), y ≤ n, isprime =
     isprime (y)}

if  isprime then
     size := size + 1 ;
     primes (size) := y
fi ;
{primes = primes (y+1) = primes (y), odd (y), y ≤ n}
 y := y + 2
{primes = primes (y-1), odd (y), y ≤ n + 2}
```

The final program with its intermediate assertions as documentation is shown in Figure 3.3.

3.4 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:

(3.4.1) primes = [[p | p ≤ n, p is prime]]

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:

(3.4.2) primes = [[2]] U oddprimes (3, n)

using the functions:

FIGURE 3.3

AXIOMATIC SOLUTION OF PRIMES PROBLEM

```
{ n ≥ 2 }
 primes (1) := 2 ;  size := 1 ;
{primes  =  primes (2)}
 y  :=  3 ;
{primes = primes (y-1), odd (y),  y ≤ n + 2}
 while  y ≤ n  do
     isprime := true ;  j := 2 ;
     {primes = primes (y-1), odd (y), 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),  y ≤ n,
          isprime = isprime (y)}
     if  isprime  then
          size :=  size + 1 ;
          primes (size) :=  y
     fi ;
     {primes = primes (y) = primes (y + 1), odd (y), y ≤ n}
          y  :=  y + 2
     {primes = primes (y - 1), odd (y), y ≤ n + 2}
od
{primes  =  primes (n)}
```

$$\underline{\text{oddprimes}}\ (\ell,u) = [[\ell \mid \underline{\text{isprime}}\ (\ell)]] \cup \underline{\text{oddprimes}}\ (\ell+2,u) \quad \underline{\text{if}}\ \ell \leq u$$

$$= \underline{\text{empty}} \quad \underline{\text{if}}\ \ell > u$$

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

The functional specification (3.4.2) 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.

(3.4.3) [primes := [[2]] U $\underline{\text{oddprimes}}$ (3, n_{in})]

primes (1) := 2 ; size := 1 ;

y := 3 ;

[primes := primes_{in} U $\underline{\text{oddprimes}}$ (y_{in}, n_{in})]

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

(3.4.4) [primes := primes_{in} U $\underline{\text{oddprimes}}$ (y_{in}, n_{in})]

$\underline{\text{while}}$ $y \leq n$ $\underline{\text{do}}$

[primes := primes_{in} U [[y_{in} | $\underline{\text{isprime}}$ (y_{in})]]

y := y_{in} + 2]

$\underline{\text{od}}$

As with all expansions, the associated lemmas given in Table 3.2 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.

step	primes	y
initially	$primes_{in}$	y_{in}
loop body	$primes_{in}$ U [[y_{in} | <u>isprime</u> (y_{in})]]	y_{in} + 2
loop function	$primes_{in}$ U [[y_{in} | <u>isprime</u> (y_{in})]] U <u>oddprimes</u> (y_{in} + 2, n_{in}) = $primes_{in}$ U <u>oddprimes</u> (y_{in}, n_{in})	--

Since the final value for primes is the same as the intended function of (3.4.4), this case is proved.

3. Whenever y > n, is the intended function of the loop an identity? Yes, since y > n, the set <u>oddprimes</u> (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 (3.4.4).

(3.4.5)
```
[primes := primes_in U [[y_in | isprime (y_in)]] ,
    y := y_in + 2]
  [isprime := isprime (y_in)]
  if isprime then
       size := size + 1 ;
       primes (size) := y
```

```
        fi ;
        y := y + 2
```

The final expansion is the loop necessary to calculate isprime.

```
(3.4.6)  [isprime := isprime (y_in)]
          isprime := true ;   j := 2 ;
         while   j ≤ size  and  isprime  do
                 isprime := (y  rem  primes (j) ≠ 0) ;
                 j := j + 1
         od
```

The complete program with its intermediate functions as documentation is given in Figure 3.4.

FIGURE 3.4

FUNCTIONAL SOLUTION OF PRIMES PROBLEM

```
[primes := [[2]] U oddprimes (3, n)]
    primes (1) := 2 ;   size := 1 ;
    y := 3;
    [primes := primes_in U oddprimes (y_in , n_in)]
        while y ≤ n do
            [primes := primes_in U [[y_in | isprime (y_in)]]
             y := y_in + 2]
                [isprime := isprime (y_in)]
                    isprime := true ;    j := 2 ;
                    while j ≤ size and isprime do
                        isprime := (y rem primes (j)
                                   ≠ 0) ;
                        j := j + 1
                    od ;
                if isprime then
                    size := size + 1 ;
                    primes (size) := y
                fi ;
                y := y + 2
        od
```

4. 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.

4.1 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 runtime). 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. The techniques are applicable in a stepwise manner, at various levels in the correctness and development process, dealing with only small segments of code, 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. 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, 1974]. Some results have recently been published in generating intended loop functions [Blikle, 1977].

4.2 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 2. 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 the appropriate intended function. In illustration, consider the following simple program:

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

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

{ true }	[I = 2]
I := 1	[I = 1]
{I = 1}	I := 1
I := I + 1	[I = I_{in} + 1]
{I = 2}	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, e.g., the statement I := I + 1 is defined by the function [I = I_{in} + 1] (shorthand for f = [[((I_{in}, 1), I) | I = I_{in} + 1]]) and this is true independent of where that statement is imbedded in the program. It defines the effect of that statement in a variable-free way, i.e., I represents the output value space of the function and I_{in} and 1 represent the input value space.

The variable free aspect of the functional model versus the variable dependent aspect of the axiomatic model is demonstrated by their model of the assignment statement. Consider the axiomatic rule for assignment given in Figure 2.2. In the assignment x := f, if f is any expression not involving the program variable x then there can be no way for the post assertion Q to capture the old value of x. Thus, in the multiplication example, in order to assert that Z = A * B, neither of the values of the input variables A or B can be destroyed. Since the algorithm used destroys the value of B, a copy of this value (Y) must be made in order to prove correctness. This is so the relationship can be written in an invariant way, even though the value

of Y is changing dynamically. The output assertion requires the variable B be unchanged as a reference point.

In contrast, the functional model handles assignment "naturally" since a function, like an assignment, is a mapping of input values to output values. The functional specification is not in terms of the variables at all but the values of the variables; i.e., Z and Zin represent different value sets of the same program variable, at point of input to the program segment specified and at point of exit. The function gives the relationship between the two value sets.

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.

This cannot be done in the axiomatic model. An assertion about a variable depends on the history of the use of that variable and on its interdependence on other variables. In addition, assertions contain global information about nonlocally affected variables. Thus changes in a program

affecting a particular variable are not necessarily independent of the remainder of the program and will usually require new proofs of all assertions containing that variable.

Bottom Up Correctness - An added effect of this difference is that given a program without any functional specifications, the intended function for any prime program can be defined depending only on the subhierarchy of the prime program, i.e., functional correctness can be approached bottom up. Suppose the functional correctness of the program in Figure 4.1 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 in

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

or

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

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

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

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

FIGURE 4.1

PROGRAM TO COMPUTE I = N

```
{N ≥ 0}
 I := 0
{I ≤ N}
while I < N do
      I := I + 1
od
{I = N}
```

variable I as well as its relationship to the nonlocally referenced N. Suppose, however, the straightforward approach adopted for the functional method was tried:

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

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

$$\{P \wedge B\} \quad I := I + 1 \quad \{P\}$$

or

$$\{I \leq N, \quad I < N\} \quad I := I + 1 \quad \{I \leq N\}.$$

Thus, without considering the loop as a whole, there is no practical way of determining the correct loop body post-condition, that is, there is no practical way to assure that the choice of the post-condition 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 Figure 4.1, given the program as a whole, the intended function of the loop is actually

$$[\ I = N_{in} \qquad \text{if } I_{in} \leq N_{in}\ ].$$

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 ($N \geq 0$) instead of the entire input domain (N an integer).

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. For example, consider the problem of finding the intended function of a binary search program if you do not know the input array is ordered.

4.3 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 (xin) is the loop invariant [Mills, 1975]. In the multiplication example, the functional specification for the loop is Z := Zin + Ain * Bin thus the loop invariant is

f (x) = f (xin)

Z + A * B = Zin + Ain * Bin

Z = Zin + Ain * Bin = A * B

Z = Zin + A * (Bin - B), since A = Ain

Z = A * (Bin - B), since Zin = 0 from the initial assertion and the composition of the statements preceding the loop

In the program in Figure 2.3, the variable B plays the role of Bin and the variable Y plays the role of B.

5. 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.

It is not clear which approach is more effective in an operational environment. It may, in fact depend upon the particular environment, and the problems that arise. Enough is still not known about the kinds of errors designers and programmers make in different environments and therefore whether a variable-free or variable-dependent approach is best. One possibility is for the developer to be aware of both and use both in the development of programs. Certainly, one could be used in formally deriving the program and the other 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. C. A. R. Hoare. An axiomatic basis for computer programming. CACM, 12 (October 1969), pp. 576-583.

2. H. D. Mills. The new math of computer programming. CACM, 18 (January 1975), pp.

3. B. Wegbreit. The synthesis of loop predicates. CACM, 17 (February 1974), pp. 102-112.

4. E. W. Dijkstra. A Discipline of Programming. Prentice-Hall (1976).

5. J. S. Gishen and R. E. Noonan. Toward a methodology for the formal derivation of programs. IEEE Transactions on Software Engineering (to appear).

6. A. Blikle. An analytic approach to the verification of iterative programs. Information Processing 77, (1977), pp. 285-290.

7. H. D. Mills. Mathematical foundations for structured programming. IBM Federal Systems Division, FSC 72-6012 (1972).

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-78-0737	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A COMPARISON OF THE AXIOMATIC AND FUNCTIONAL MODELS OF STRUCTURED PROGRAMMING.	5. TYPE OF REPORT & PERIOD COVERED: Interim rept.
	6. PERFORMING ORG. REPORT NUMBER: Tech Rpt TR-630
7. AUTHOR(s): Victor R. Basili, Robert E. Noonan	8. CONTRACT OR GRANT NUMBER(s): AFOSR-77-3181
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Maryland, Department of Computer Science, College Park, MD 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304 A 2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, DC 20332	12. REPORT DATE: Feb 78
	13. NUMBER OF PAGES: 40
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

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

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

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409 022