Mathematical Principles for a First Course in Software Engineering

HARLAN D. MILLS, SENIOR MEMBER, IEEE, VICTOR R. BASILI, SENIOR MEMBER, IEEE, JOHN D. GANNON, AND RICHARD G. HAMLET, MEMBER, IEEE

Abstract—**The discipline of software engineering has transferred the commonsense methods of good programming and management to large software projects. It has been less successful in acquiring a solid theoretical foundation for these methods. We have developed an introductory computer science course, much as calculus is a basic course for mathematics and the physical sciences, concerned primarily with theoretical foundations and methodology rather than apprenticeship through applications. This paper describes the principles taught in the course and gives a small example illustrating them.**

Index Terms—**Formal specification, programming language semantics, program verification, software development.**

I. Introduction

SOFTWARE engineering is the name given to the art of programming (and surrounding activities) when the art is replaced by a discipline using well-defined methods and formal skills. During the last two decades, a great deal has been learned about good programming practices: structured programming [1], [14], information hiding [12], and data abstraction [7]. The spread of this knowledge beyond expert programmers can be credited to software engineering. The transfer of knowledge to routinely trained technicians, the codification of common sense, and the introduction of management control are certainly functions proper to engineering, and software engineering has accomplished these things for programming.

The success and growth of any engineering discipline has never rested entirely on the organization of trial-and-error knowledge, however. Application of deep theoretical results is also required to progress beyond the initial success that spreading common sense brings. The role of the engineer is sometimes to invent the required theory; more often, it is only to apply an idea from a more abstract discipline to a problem the engineer understands. Furthermore, the application must meet a requirement peculiar to engineering: it must be in a form that can be used to solve practical problems.

The work of [2], [3], [5], [6], [8], and [11] comprises a theoretical framework for programming based on mathematical foundations. This work continues to inspire advances in formal specification, programming language semantics, and program verification. Because our understandings of these topics are quite recent, people regard them as subjects fit for graduate classes. Our belief is that the time has come for computer science students to start out with these ideas, not end up with them. At their roots, these deep simplicities can be formulated in discrete mathematics. This is material that undergraduates can learn easily because understanding it does not require a wide context of programming experience. By teaching principles of syntax, semantics, correctness, and abstraction, we arm the student with solution patterns so that program design becomes a problem-solving process with new-found mental tools in a new domain.

We have developed a two-semester course for software engineering [9], much as calculus is a basic course for mathematics and the physical sciences, concerned primarily with methodology rather than subject matter. In fact, we introduce a "program calculus" that deals with the functions computed by programs. Just as for ordinary calculus, there are two main problems in the program calculus. First, given a program, find its meaning (analogous to a function's derivative), and second, given a meaning, find a program with that meaning (analogous to a function's integral). This ability to derive functions from programs in the program calculus is of great value in computer science and in engineering as well. First, it permits a mathematical treatment of program correctness, namely, whether a program specifies correct behavior of the computer for every possible input. But even more importantly, it leads to a systematic design discipline for writing programs that are correct to begin with and which do not require debugging.

II. Topics for the Introductory Course

In this section we outline the major principles covered in the course. They represent a synthesis of an integrated theoretical foundation for programming. They can be characterized by a mathematical formalism, simplified to the level necessary for the problem at hand, covering a large piece of the programming domain in a consistent way and permitting the process to be applied to larger problems.

The simplicity and generality of the formalism permit the material to be taught to beginning college students. It

Manuscript received July 15, 1986; revised August 25, 1988. This work was supported in part by the U.S. Air Force Office of Scientific Research.

H. D. Mills is with the Department of Computer and Information Science, University of Florida, Gainesville, FL 32601.

V. R. Basili and J. D. Gannon are with the Department of Computer Science, University of Maryland, College Park, MD 20742.

R. G. Hamlet is with the Department of Computer Sciences, Portland State University, Portland, OR 97207.

IEEE Log Number 8926730.

forms a basis for their understanding of the programming process and product and acts as a mechanism for communication.

A. Programming Methods

During the first semester, programs are developed using stepwise refinement, while in the following semester systems of programs and data are constructed using data abstraction. Programming has two distinct phases:

1) *design*, thinking out what the program should be in order to solve the problem, and

2) *development*, putting the program text in execution form.

In the stepwise refinement of a program, text designed to carry out a task in more detail is called a *design part.* A design part may itself contain more detailed task descriptions (in the form of comments) to be carried out by additional design parts. The result of the design phase will be a hierarchy of design parts which collectively solve the problem at hand. Each design part is a statement typically with 5–15 lines, perhaps with 2–4 remaining tasks to be designed at the next level.

After the program has been entirely refined into a hierarchy of design parts, the translation into machine-readable form begins. A sequence of executable programs, each reflecting a larger part of the design, can facilitate orderly and systematic translation into Pascal. Each program in such a sequence is called a *development program.* Development programs are accumulations of design parts, which grow in size until the entire design has been turned into Pascal. Each development program is defined so that it can be executed and tested to verify correct translation at each step of development.

In practice, the translation of a design into Pascal can be done in chunks larger than one design step, typically 15–50 lines at a time. That is, each successive development program is created by combining a few more design parts with the last tested development program. Functional testing methods are used to verify correct execution of development programs after adding temporary statements to create visible output.

An *abstract data type* is a collection of operations and data declarations defined so that the operations are the only means of accessing the data. Since they access data objects indirectly through operations, users of a type are unaffected by changes to the type's data declarations (which might be made to improve functionality or efficiency). Abstract data types represent potentially reusable modules and should be tested independently of their uses in programs. In testing a data type, legal combinations of its operations are applied to representative objects.

B. Programming Languages

Programs are written in three increasingly complex subsets of the programming language Pascal. In the simplest subset, CF Pascal, there is but a single kind of data (characters) and a single data structure (files of characters that can only be accessed sequentially). Restricting our attention to so simple a language emphasizes program design rather than language features.

Small, but classical, problems lead to interesting program design problems in CF Pascal. For example, breaking a text file into lines of a given length (specified by the length of a file of blanks since there are no integers) cannot be done without a working sense of abstraction. Sorting and reversing files in $n^* \ln (n)$ time are also challenges. Consider adding two 100-digit numbers in different files and writing the result to a third file. The input files are read left to right, but digits must be added, and carries must be computed from right to left. It is easy to see that the problem requires one pass over the two files for the add and carry logic, but three file reverses. With an n^2 reverse, the solution will execute in $n + 3n^2$ time where it only takes n for what seemed the hard part. So reducing $n + 3n^2$ to $n + 3n^* \ln (n)$ by finding an $n^* \ln (n)$ reverse becomes an interesting problem. CF Pascal is a teacher's helper in a real sense—an austere tool that rewards good programs in a visible way.

The second language subset, D Pascal, permits the same functions to be created with smaller and simpler programs than are possible in CF Pascal. D Pascal also contains language features (type declarations and records) needed to implement abstract data types. The final Pascal subset, O Pascal, introduces control structures and data types to help optimize programs by providing random access to statements (gotos) and data (arrays). O Pascal language features should be used only when they are needed for algorithm optimization and when their functions can be determined and verified at least informally.

C. Mathematical Basis

The entire mathematical basis for the program calculus rests on just five discrete mathematical structures of character data: strings, lists, sets, relations, and functions. These five structures are not only sufficient to deal with program correctness and program design, but also admit treatment at various levels of formality with a mixture of English and mathematical notation. Some sets are more easily and precisely described in English than in mathematics, but are sets no less because of the mode of their description. Many programming problems are better stated in English than mathematics, and we need to be able to treat questions of program correctness and design independently of the mode of description.

Understanding a program as a mathematical object is understanding the functional behavior it induces in a computer. An *execution state* is a relation or function whose domain is the identifiers of a program and whose range is the values attached to those identifiers. The semantic meaning of a program will be a mathematical relation or function, a set of ordered pairs of states, that defines a correspondence between an input state and an output state.

D. Meanings of Program Parts

It is convenient to have a notation for meaning relations or functions, and we adopt a convention similar to one used by Kleene: the meaning function corresponding to a program object is denoted by a box around that object. The meaning of an identifier $V1$ in execution state s is simply the value of $V1$ in the state:

$$\boxed{V1}(s) = s(V1).$$

Values of literal character expressions do not depend on the execution state at all. The meaning of an assignment statement is a function from execution states to execution states. The intuitive meaning of the assignment statement as an execution state transformation is that the identifier on the left-hand side ceases to be associated with an old value and becomes associated with a new value, obtained from the expression on the right-hand side:

$$\boxed{V1 := V2} = \{<r, s>: s \text{ is the same as } r \text{ except that } \boxed{V1}(s) = \boxed{V2}(r)\}.$$

The meaning of an IF statement with Boolean condition b and statements t and e is

$$\boxed{\text{IF } b \text{ THEN } t \text{ ELSE } e} = \{<s, \boxed{t}s>: \boxed{b}(s)\} \cup \{<s, \boxed{e}s>: \neg\boxed{b}(s)\}.$$

The first set contains all state pairs in which the condition b holds, and the second set contains those pairs in which the condition does not hold. There is no evaluation of these sets in some order. They simply contain or fail to contain certain pairs.

The meaning of a WHILE statement with Boolean condition b and statement d is defined recursively:

$$\boxed{\text{WHILE } b \text{ DO } d} = \boxed{\text{IF } b \text{ THEN BEGIN } d\text{; WHILE } b \text{ DO } d \text{ END}}.$$

The right-hand side of this definition can be rewritten as the composition of two functions:

$$\boxed{\text{WHILE } b \text{ DO } d} = \boxed{\text{IF } b \text{ THEN } d} \circ \boxed{\text{WHILE } b \text{ DO } d}.$$

This recurrence equation has solutions (the meaning of the WHILE statement) which can be checked by substitution. The WHILE statement verification rule [9] states that a function f is the meaning of a WHILE statement if and only if it satisfies three conditions.

1) f and the WHILE statement have identical domains. (The domain of a WHILE statement is the set of states for which the WHILE statement terminates.)

2) Restricting f's domain to those states in which b is false yields an identity function.

3) $f = \boxed{\text{IF } b \text{ THEN } d} \circ f$.

E. Determining the Meaning of Program Parts

A *concurrent assignment* summarizes the effects of several statements, mapping one state to another. A list of variables is written on the left-hand side of the assignment operator, and a list of expressions is written on the right-hand side, these two lists being of equal length. The expressions, computed all at the same time, are the values taken by the corresponding variables.

Conditional assignments can be defined recursively by the following rules.

1) A concurrent assignment is a conditional assignment.

2) If b is a Boolean condition and c is a conditional assignment, then $(b \rightarrow c)$ is a conditional assignment.

3) If b is a Boolean condition and c, d are conditional assignments, then $(b \rightarrow c) \mid d$ is a conditional assignment.

For state s, the meaning of a conditional assignment of the form

$$(b_1 \rightarrow c_1) | (b_2 \rightarrow c_2) | \cdots | (b_n \rightarrow c_n)$$

for any number of Boolean conditions (b_i) and conditional assignments (c_i) is the meaning of the first conditional assignment, say c_k, such that

1) all Boolean conditions before b_k have the value false in state s, and

2) Boolean condition b_k has the value true in state s.

The meaning is undefined for state s if any of the following occur:

1) none of the Boolean conditions evaluates to true in s,

2) the first Boolean expression that does not evaluate to false is undefined for s, or

3) c_k is undefined for s.

For example, the meaning of the statement

```
BEGIN
  V1 := V2;
  V2 := V3;
  IF V1 < V2 THEN V3 := V1 ELSE V3 := V2
END
```

can be expressed as the conditional assignment

$$(V2 < V3 \rightarrow V1, V2, V3 := V2, V3, V2) |$$
$$(V2 \geq V3 \rightarrow V1, V2 := V2, V3).$$

Symbolic execution is a method of tracing the values of variables through execution using only their names, not particular values. A *trace table* is a systematic method for carrying out symbolic execution. A trace table has a row for each statement that is executed and a column for each variable that might acquire a new value during execution. The "values" in a trace table are expressions, and the rows keep track of current expressions in terms of the original starting expressions. A *conditional trace table* is a trace table with an additional column of conditions, namely, those required for the assignments in the table to take place. For example, the BEGIN statement above will use one of two sequences of assignments, namely,

$$(V1 := V2;\ V2 := V3;\ V3 := V1) \quad \text{or}$$
$$(V1 := V2;\ V2 := V3;\ V3 := V2),$$

depending on whether the THEN or ELSE part of the IF statement is selected during execution. Each sequence can be handled by a separate conditional trace table. The table for the case when the THEN part is executed is shown below.

Statement	Condition	$V1$	$V2$	$V3$
$V1 := V2$		$V2$		
$V2 := V3$			$V3$	
IF $V1 < V2$	$V2 < V3$			
THEN $V3 := V1$				$V2$

Each row of the table shows values in terms of the original variables. The condition $V2 < V3$ in the third row is the value of $V1 < V2$ because at that point $V1$ has the original value of $V2$, and $V2$ has the original value of $V3$ (obtained from the row above). The net result for this conditional trace table is a conditional assignment:

$$(V2 < V3 \rightarrow V1,\ V2,\ V3 := V2,\ V3,\ V2).$$

Similarly, the table for the ELSE part derives the second part of the conditional assignment above.

F. Program Correctness

Given a program specification relation r and a program P, we say that P is correct with respect to r if, for every member x of the domain of r (an instance of input data), P produces some member of the range of r which corresponds to x. That is, for each input x, P produces result y such that $<x, y> \in r$. What P does to input data not in the domain of r is not important since r should define all behavior important to the problem solver. A simplifying condition for demonstrating that a program satisfies its specification is given in the following theorem, called the Correctness Theorem [9].

Correctness Theorem: Program P is correct with respect to specification relation r if and only if domain $(r \cap \boxed{P}) = \text{domain}(r)$. Note that, if r is a function f, then the following corollary holds.

Correctness Corollary: Program $\boxed{P}$ is correct with respect to specification function f if and only if $f \cap \boxed{P} = f$, that is, if and only if $f \subseteq \boxed{P}$.

G. Data Abstraction Correctness

The essence of data abstraction is captured by a diagram showing the relationship between the *concrete* world objects manipulated by Pascal procedures (e.g., P) and the *abstract* world objects the programmer uses in abstract operations (e.g., m) to achieve a solution. A mapping must be defined between the values of concrete objects and the values of the corresponding abstract objects. We call this the *representation mapping* and, for any type, denote it A_{type}. By convention, for objects common to the concrete and abstract worlds, the representation mapping is identity. Thus, the representation mapping carries any concrete state to an abstract state.

$$\begin{tikzcd}
\{\text{abstract states}\} \arrow[r, "m"] & \{\text{abstract states}\} \\
\{\text{concrete states}\} \arrow[u, "A_{\text{type}}"] \arrow[r, "\boxed{P}"] & \{\text{concrete states}\} \arrow[u, "A_{\text{type}}"']
\end{tikzcd}$$

Intuitively, an implementation is correct if its data objects are manipulated in such a way that the abstract objects to which they correspond appear to be transformed according to the abstract operations. That is, correct implementation uses the concrete procedures and data, but in a way that mirrors the abstraction. To decide if this property holds, we must show that the diagram commutes:

$$A_{\text{type}}\ o\ m \subseteq \boxed{P}\ o\ A_{\text{type}}.$$

Of course, abstract operations like m do not really exist except in users' minds. Pascal procedures implementing abstract operations are written with two sets of comments, labeled "abs" and "con." The "abs" comments are added to modules so that users, those in the abstract world, need not examine the code (or even the "con" comments that document it). The "abs" comments replace the abstract operations in demonstrations that diagrams commute. If the implementation has been done properly, the abstract comment can be believed and used in proofs at the abstract level [4].

III. A Partial Example

In this section, we carry out part of an example that illustrates both stepwise refinement and data abstraction. The example, which prints a list of prime numbers, is large enough to require the judicious use of formal proofs, embedded in a broader activity of informal reasoning and design. In particular, the example illustrates that stepwise refinement permits the control of details, step by step, in both data and operations. A lower-level specification uses only variables from the design it supports rather than those introduced for its implementation. This property of deferring details while maintaining control of the design is essential in scaling up methods of "programming in the small."

A. Developing and Proving the Solution

This section illustrates a typical design step in a small problem, printing the prime numbers P such that $2 \leq P \leq N$. The solution prints 2, constructs a list containing the remaining odd prime numbers less than or equal to N, and prints the list elements. (Lists are denoted by angle brackets and list concatenation by an ampersand (&).)

Design Part 1

```
{print the list of prime numbers P such that 2 ≤
P ≤ N}

WRITELN(2);
EmptyList(Primes);
NextP := 3;
{Primes := Primes &
           ND(NextP,NextP, N,Primes)};
{print Primes}
```

EmptyList is an operation on abstract objects of type "list of integers" that initializes a list object that contains no elements:

```
PROCEDURE EmptyList(VAR L: List);
  { abs: L := <  > }
```

ND(L, C, U, Primes) is a list of numbers between C and U that are not divisible by any of the numbers in Primes or any of the numbers between L and $C - 2$. It is defined as follows:

$$\text{ND}(L, C, U, \text{Primes}) = \begin{cases} < C: \forall P \in (\text{Primes} \,\&\, \text{ND}(L, L, C-2, \text{Primes})), \\ \quad (C \bmod P) \neq 0 > \\ \quad \&\, \text{ND}(L, C+2, U, \text{Primes}) \text{ if } L \leq C \leq U \\ < > \text{ if } L \geq C > U. \end{cases}$$

We can use a trace table to calculate the value of Primes that is printed at the end of Design Part 1.

Condition	Primes	NextP
	< >	
		3
3 ≤ 3 ≤ N	< > & ND(3, 3, N, < >)	

Assuming $N \geq 3$, we expand ND's definition to check that its behavior meets our expectations.

$$\begin{aligned} &\text{ND}(3, 3, N, < >) \\ &= (<3: \forall P \in (< > \,\&\, \text{ND}(3, 3, 1, < >)), (3 \bmod P) \neq 0> \,\&\, \text{ND}(3, 5, N, < >)) \\ &= <3> \,\&\, (<5: \forall P \in (< > \,\&\, \text{ND}(3, 3, 3, < >)), (5 \bmod P) \neq 0> \,\&\, \text{ND}(3, 7, N, < >)) \\ &= <3> \,\&\, <5: \forall P \in (< > \,\&\, <3>)), (5 \bmod P) \neq 0> \,\&\, \text{ND}(3, 7, N, < >) \\ &= <3> \,\&\, <5: \forall P \in <3>), (5 \bmod P) \neq 0> \,\&\, \text{ND}(3, 7, N, < >) \\ &= <3> \,\&\, <5> \,\&\, \text{ND}(3, 7, N, < >). \end{aligned}$$

When an odd number i is considered for membership in ND, it is divided by the odd numbers between 3 and $i - 2$ that are already members of ND. If none of the members of ND divides i, then i is also a member of ND.

The fourth step of Design Part 1 can be refined into the following design part.

Design Part 1.1

```
{Primes := Primes &
           ND(NextP,NextP, N,Primes)}
WHILE NextP <= N DO
  BEGIN
    {IsPrime := (for all members P of Primes,
                 (NextP mod P) < > 0)};
    IF IsPrime THEN Append(Primes,NextP);
    NextP := NextP + 2
  END
```

If L has not reached its maximum size, Append concatenates a singleton list containing Elt to the right end of L. Otherwise Append is an identity function.

```
PROCEDURE Append(VAR L: List; Elt: EltType);
  { abs: ( Length(L) < MaxSize →
                L := L & <Elt> ) |
         ( Length(L) ≥ MaxSize → I ) }
```

If the design is under intellectual control at this point, we need to demonstrate that the WHILE statement above (which we call W) computes its specified function. In order to do this, we first determine f, the meaning of W, and show that it has the three properties listed at the end of Section II-D.

$$\begin{aligned} f = (&\text{NextP} \leq \text{N} \rightarrow \text{Primes} := \\ &\text{Primes} \,\&\, \text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes})) \mid \\ (&\text{NextP} > \text{N} \rightarrow \text{I}). \end{aligned}$$

Then we use the Correctness Corollary from Section II-F to verify

$$\begin{aligned} (&f \cap \\ &(\text{Primes} := \text{Primes} \,\&\, \text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes}))) \\ &= f. \end{aligned}$$

This last step is easy because ND(NextP, NextP, N, Primes) = < > when NextP > N. Thus, f could be written

$$\begin{aligned} f &= (\text{NextP} \le \text{N} \rightarrow \text{Primes} := \text{Primes} \;\&\; \text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes}))\,| \\ &\quad\; (\text{NextP} > \text{N} \rightarrow \text{Primes} := \text{Primes} \;\&\; \text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes})) \\ &= (\text{NextP} \le \text{N or NextP} > \text{N} \rightarrow \text{Primes} := \text{Primes} \;\&\; \text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes}) \\ &= (\text{true} \rightarrow \text{Primes} := \text{Primes} \;\&\; \text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes}) \\ &= \text{Primes} := \text{Primes} \;\&\; \text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes}). \end{aligned}$$

To verify that f is the meaning of the WHILE statement more easily, we eliminate details introduced by purely local variables (like NextP, whose value is not needed after W terminates), and size constraints on objects (like the length of Primes).

The three steps of the proof are as follow.

1) *Domain* (f) = *domain* $(\boxed{W})$:

Domain (f) = *domain* (ND (NextP, NextP, N, Primes)): ND(NextP, NextP, N, Primes) is defined whenever the mod function within it is defined, i.e., if either $0 \notin$ Primes or $0 \notin$ ND(NextP, NextP, N − 2, Primes). The latter condition could be false only if Primes = < >, NextP = 0, and NextP < (N − 2).

Domain $(\boxed{W})$: W terminates immediately if NextP > N. If NextP ≤ N, the body of W is executed, and normal termination occurs if NextP is incremented on each execution of the body of W and if the result of the mod function is defined for all members of Primes on each iteration. The first condition is clearly true by inspection. The mod function is defined if $0 \notin$ Primes and if 0 is not one of the values of NextP appended to Primes [i.e., Primes = < >, NextP = 0, and NextP < (N − 2)].

Thus, the domains are identical.

2) *(NextP > N → f) = (NextP > N → I)*: When f's domain is restricted to those states for which the WHILE condition evaluates to false, f is an identity function.

$$\begin{aligned} &(\text{NextP} > \text{N} \rightarrow f) = \\ &\quad (\text{NextP} > \text{N and NextP} \le \text{N} \rightarrow \\ &\quad \text{Primes} := \text{Primes} \;\& \\ &\quad \text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes}))\,| \\ &\quad\quad (\text{NextP} > \text{N and NextP} > \text{N} \rightarrow I), \end{aligned}$$

which reduces to

$$\begin{aligned} &((\text{false} \rightarrow \cdots)\,|\,(\text{NextP} > \text{N} \rightarrow I)) \\ &\quad = (\text{NextP} > \text{N} \rightarrow I), \end{aligned}$$

an identity function.

3) Let IF be

```
IF NextP <= N THEN
   BEGIN
      {IsPrime := (for all members P of Primes,
      (NextP mod P) < > 0)};
      IF IsPrime THEN Append(Primes, NextP);
      NextP := NextP + 2
   END
```

Then, $\boxed{\text{IF}}$ is

$$\begin{aligned} &(\text{NextP} \le \text{N and} \\ &\quad (\forall P \in \text{Primes}, (\text{NextP mod } P) \ne 0) \rightarrow \\ &\quad \text{Primes, NextP} := \\ &\quad \text{Primes} \;\&\; <\text{NextP}>, \text{NextP} + 2)\,| \\ &(\text{NextP} \le \text{N and } \exists P \in \text{Primes}, (\text{NextP mod } P) = 0 \rightarrow \\ &\quad \text{NextP} := \text{NextP} + 2)\,| \\ &(\text{NextP} > \text{N} \rightarrow I), \end{aligned}$$

We must demonstrate $f = \boxed{\text{IF}}\, o\, f$. Trace tables provides a convenient way to study the composition, showing the effect of each part of $\boxed{\text{IF}}$ composed with each part of f.

Part	Condition	Primes	NextP
IF	NextP ≤ N and (∀ P ∈ Primes, (NextP mod P) ≠ 0)	Primes & < NextP >	NextP + 2
f	(NextP + 2) ≤ N	Primes & < NextP > & ND(NextP + 2, NextP + 2, N, Primes & < NextP >)	

This trace table computes the part function

$$\begin{aligned} &(\text{NextP} + 2) \le \text{N and } (\forall P \in \text{Primes}, (\text{NextP mod } P) \ne 0) \rightarrow \\ &\quad \text{Primes} := \text{Primes} \;\&\; <\text{NextP}> \;\&\; \text{ND}(\text{NextP} + 2, \text{NextP} + 2, \text{N}, \text{Primes} \;\&\; <\text{NextP}>). \end{aligned}$$

We need to show that < NextP > & ND(NextP + 2, NextP + 2, N, Primes & < NextP >) is identical to f in this limited domain. We calculate ND(NextP, NextP, N, Primes), but restrict its domain to those states in which the domain restriction

$$\begin{aligned} &(\text{NextP} + 2) \le \text{N and} \\ &\quad (\forall P \in \text{Primes}, (\text{NextP mod } P) \ne 0). \end{aligned}$$

has the value true.

$$\text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes}) = <\text{NextP}: \forall P \in (\text{Primes} \ \& \ \text{ND}(\text{NextP}, \text{NextP}, \text{NextP} - 2, \text{Primes})), (\text{NextP} \bmod P) \neq 0> \ \& \ \text{ND}(\text{NextP}, \text{NextP} + 2, \text{N}, \text{Primes}) \quad (1)$$

and

$$\text{ND}(\text{NextP}, \text{NextP}, \text{NextP} - 2, \text{Primes}) = < >. \quad (2)$$

Substituting (2) into (1) yields

$$\text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes}) = <\text{NextP}: \forall P \in \text{Primes}, (\text{NextP} \bmod P) \neq 0> \ \& \ \text{ND}(\text{NextP}, \text{NextP} + 2, \text{N}, \text{Primes}). \quad (3)$$

In the domain $\forall P \in \text{Primes}, (\text{NextP} \bmod P) \neq 0$, (3) can be rewritten as

$$\text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes}) = <\text{NextP}> \ \& \ \text{ND}(\text{NextP}, \text{NextP} + 2, \text{N}, \text{Primes}). \quad (3')$$

Thus, our two original terms are identical if

$$\text{ND}(\text{NextP} + 2, \text{NextP} + 2, \text{N}, \text{Primes} \ \& <\text{NextP}>) = \text{ND}(\text{NextP}, \text{NextP} + 2, \text{N}, \text{Primes}).$$

Both these terms represent sequences of numbers between NextP + 2 and N that are tested to determine if they are divisible by any number in another sequence of numbers. The term on the right-hand side of the equation checks divisors in the range NextP + 2..NextP + 2 − 2 and the members of Primes & < NextP >. The term on the left-hand side checks divisors in the range NextP..NextP + 2 − 2 and the members of Primes. Thus, the terms must be identical, and the part function can be rewritten as

$$(\text{NextP} + 2) \leq N \text{ and } (\forall P \in \text{Primes}, (\text{NextP} \bmod P) \neq 0) \rightarrow \text{Primes} := \text{Primes} \ \& \ \text{ND}(\text{Next}P, \text{NextP}, \text{N}, \text{Primes}).$$

The next case resulting from this composition is as follows.

Part	Condition	Primes	NextP
IF	NextP ≤ N and (∀ P ∈ Primes, (NextP mod P) ≠ 0)	Primes & < NextP >	NextP + 2
f	(NextP + 2) > N	Primes & < NextP > & ND(NextP + 2, NextP + 2, N, Primes & < NextP >)	

This trace table computes the part function

$$(\text{N} - 2) < \text{NextP} \leq \text{N and } (\forall P \in \text{Primes}, (\text{NextP} \bmod P) \neq 0) \rightarrow \text{Primes} := \text{Primes} \ \& <\text{NextP}> \ \& < >.$$

In the domain (N − 2) < NextP ≤ N where no member of Primes divides NextP evenly,

$$\text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes}) = (<\text{NextP}> \ \& < >),$$

so this part function could be combined with the previous part function to obtain

$$\text{NextP} \leq \text{N and } (\forall P \in \text{Primes}, (\text{NextP} \bmod P) \neq 0) \rightarrow \text{Primes} := \text{Primes} \ \& \ \text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes}).$$

We need to perform four more function compositions to obtain the rest of the function:

$$(\text{NextP} \leq \text{N and } (\exists P \in \text{Primes}, (\text{NextP} \bmod P) = 0) \rightarrow \text{Primes} := \text{Primes} \ \& \ \text{ND}(\text{NextP}, \text{NextP}, \text{N}, \text{Primes})) \mid (\text{NextP} > \text{N} \rightarrow I),$$

which, combined with the already-computed parts, yields a function that is identical to f.

B. Verifying the Data Abstraction

In refining the remainder of the loop body, additional operations must be added to the data type so that the values of the elements in the list can be obtained sequentially without modifying the list value. The abstract type used to represent Primes is "list of integers with a reading pointer." The operations of this type are EmptyList, Append, Head, and Next. Head(Primes, Trial) assigns Trial the value of the first element in Primes (if Primes is not empty) and the value EndList otherwise. In addition, Primes is prepared for reading, which we indicate by writing the elements in Prime as

< already-read values of Primes >
& < to-be-read values of Primes >.

Next(Primes, Trial) assigns the next value to be read in Primes to Trial and advances the reading pointer. If no more values remain to be read, Next assigns to Trial the value EndList.

A representation for list objects and its representation function must be chosen.

```
TYPE
EltType = Endlist..MAXINT;
List = RECORD
          V: ARRAY [1..MaxSize] OF EltType;
          Size, Current: 0..MaxSize
       END;
```

The representation function A_{List} is a schema that specifies how a list object L is mapped.

$\{(s, t)$: $0 \leq L.\text{Current}(s) \leq L.\text{Size}(s) \leq \text{MaxSize}$, t is the same state as s except t contains a new pair $(L, L(t))$ with

$$L(t) = <L.\text{V}(s)[1], \cdots, L.\text{V}(s)[L.\text{Current}(s)]> \ \& <L.\text{V}(s)[L.\text{Current}(s) + 1], \cdots, L.\text{V}(s)[L.\text{Size}(s)]>$$

and t contains no members whose first elements are L.V, L.Current, or L.Size

Finally, implementations are written for each operation. However, only the implementation of Head is given. Each implemented operation comes with two comments describing its function. The users' expectations can be captured by writing comments about the procedure part functions in the abstract state. These comments are labeled "abs" in the code. Of course, the concrete procedure themselves are concrete-state mappings and have part functions in that world as usual; these are labeled "con."

```
PROCEDURE Head(VAR L: List; VAR Result: EltType);
   { abs: (L = < > → Result := EndList) |
          (L = < L1, · · · , Li − 1> & < Li, · · · , Ln> →
                L, Result := < L1> & < L2, · · · , Ln>, L1)
     con: (L.Size > 0 → L.Current, Result := 1, L.V[1]) |
          (L.Size ≤ 0 → L.Current; Result := 1, EndList) }
   BEGIN {Head}
      IF L.Size > 0
      THEN BEGIN L.Current := 1; Result := L.V[1] END
     ELSE Result := EndList
   END; {Head}
```

The correspondence between the body of Head and its concrete comments is apparent from the meaning of the IF statement. We need to demonstrate

$$A_{\text{List}}\ o\ \text{Head}_{\text{abs}} \subseteq \text{Head}_{\text{con}}\ o\ A_{\text{List}}.$$

Again, trace tables prove to be a useful tool for computing function compositions. The first trace table works out the composition of the representation function and the first part of the abstract comment. In the interest of space, Current and Size have been abbreviated as C and S, respectively.

FIRST PART $A_{\text{List}}\ o\ \text{Head}_{\text{abs}}$

Condition	L	Result
0 ≤ L.C ≤ L.S ≤ MaxSize	< L.V[1], · · · , L.V[L.C] > & < L.V[L.C + 1], · · · , L.V[L.S] >	
< L.V[1], · · · , L.V[L.C] > & < L.V[L.C + 1], · · · , L.V[L.S] > = < >		EndList

The condition evaluates to true when

$$< \text{L.V}[1], \cdots, \text{L.V}[\text{L.C}] > \; = \; < > \text{ and}$$
$$< \text{L.V}[\text{L.C} + 1], \cdots, \text{L.V}[\text{L.S}] > \; = \; < >.$$

Picking L.S and L.C to be 0 achieves this result. Thus, the function is

$$\text{MaxSize} \geq 0 \text{ and L.S} = 0 \text{ and L.C} = 0 \rightarrow$$
$$\text{L, Result} := \; < >, \text{EndList}.$$

The second trace table calculates the composition of the representation function and the second part of the abstract comment.

SECOND PART $A_{\text{List}}\ o\ \text{Head}_{\text{abs}}$

Condition	*L*	Result
0 ≤ L.C ≤ L.S ≤ MaxSize	< L.V[1], · · · , L.V[L.C] > & < L.V[L.C + 1], · · · , L.V[L.S] >	
< L.V[1], · · · , L.V[L.C] > & < L.V[L.C + 1], · · · , L.V[L.S] > = < L1, · · · , Li − 1 > & < Li, · · · , Ln >	< L1 > & < L2, · · · , Ln >	L1

This condition is true when $i \geq 1$ (i.e., L.S ≥ L.C ≥ 1) and L.V[1] = L1, · · · , L.V[L.S] = Ln. Thus, the function is

$$\text{MaxSize} \geq \text{L.S} \geq \text{L.C} \geq 1 \rightarrow \text{L},$$
$$\text{Result} := \; < \text{L1} > \& < \text{L2}, \cdots, \text{Ln} >, \text{L1}.$$

The composition $A_{\text{List}}\ o\ \text{Head}_{\text{abs}}$ yields

$$(\text{MaxSize} \geq 0 \text{ and L.S} = 0 \text{ and L.C} = 0 \rightarrow$$
$$\text{L, Result} := \; < >, \text{EndList}) \;|$$
$$(\text{MaxSize} \geq \text{L.S} \geq \text{L.C} \geq 1 \rightarrow$$
$$\text{L, Result} := \; < \text{L1} > \& < \text{L2}, \cdots, \text{Ln} >, \text{L1}).$$

The next two trace tables compute the right-hand side of the equation $\text{Head}_{\text{con}}\ o\ A_{\text{List}}$.

FIRST PART $\text{Head}_{\text{con}}\ o\ A_{\text{List}}$

Condition	*L*	*L.C*	Result
L.S > 0		1	L.V[1]
0 ≤ 1 ≤ L.S ≤ MaxSize	< L.V[1], · · · , L.V[L.C] > & < L.V[L.C + 1], · · · , L.V[L.S] >		

Thus, the function is

$$\text{MaxSize} \geq \text{L.S} \geq 1 \rightarrow$$
$$\text{L, Result} :=$$
$$< \text{L.V}[1] > \& < \text{L.V}[2], \cdots,$$
$$\text{L.V}[\text{L.S}] >, \text{L.V}[1].$$

SECOND PART $\text{Head}_{\text{con}}\ o\ A_{\text{List}}$

Condition	*L*	*L.C*	EndList
L.S ≤ 0			EndList
0 ≤ L.C ≤ L.S ≤ MaxSize	< L.V[1], · · · , L.V[L.C] > & < L.V[L.C + 1], · · · , L.V[L.S] >		

Since the condition requires both L.S ≤ 0 and L.S ≥ 0, it must be the case that L.S = 0, L.C = 0, and the list

$$< \text{L.V}[1], \cdots, \text{L.V}[\text{L.C}] >$$
$$\& < \text{L.V}[\text{L.C} + 1], \cdots, \text{L.V}[\text{L.S}] >$$

must be empty. Thus, the function is

$$\text{MaxSize} \geq 0 \text{ and } L.S = 0 \text{ and } L.C = 0 \rightarrow$$
$$L, \text{Result} := <\ >, \text{EndList}.$$

The composition $\text{Head}_{\text{con}}\ o\ A_{\text{List}}$ yields

$$(\text{MaxSize} \geq L.S \leq 1 \rightarrow$$
$$L, \text{Result} :=$$
$$<L.V[1]> \& <L.V[2], \cdots,$$
$$L.V[L.S]>, L.V[1]) \mid$$
$$(\text{MaxSize} \geq 0 \text{ and } L.S = 0 \text{ and } L.C = 0 \rightarrow$$
$$L, \text{Result} := <\ >, \text{EndList}).$$

This composition yields the same results as the previous composition, but on a slightly larger domain since it places no restriction on the initial value of L.C.

C. Testing the Result

Once the design phase is complete, we switch to the development process. In this stage, abstract data types are tested independently, and design parts are incrementally assembled into execution form and tested. For example, Design Parts 1 and 1.1 in Section III-A would be assembled, and later design parts that refine

```
IsPrime := (for all members P of Primes,
(NextP mod P) < > 0);
```

might be replaced by

```
IsPrime := true;
```

which permits a development program to be tested (although in this case all odd numbers would be added to Primes). This development program would be tested with various values of N (including at least odd and even values).

To test the data abstraction functionally, we should consider the possible logical sequences of the operations EmptyList, Append, Head, and Next. For example, to check valid combinations of operations we might

1) build a list,
2) sequence through a list,
3) go back to the head of the list after partially sequencing through the list (after a Head or a Next),
4) append an element to the list while sequencing through it (after a Head or a Next), or
5) empty the list after sequencing through or building it.

These operations should be considered with at least three different kinds of list values: an empty list, a partially full list, and a full list (since we have chosen an array implementation). Some other combinations of operations are not legal and are not tested. For example, applying the Next operation to a list without executing the Head operation first will cause an error.

To check that we have covered all cases, we should consider all combinations of operations, noting whether they are legal or illegal combinations and whether the combination appears in the finished program.

Case	Operations	Legality	Used
1	EmptyList; EmptyList	Legal	No
2	EmptyList; Append	Legal	Yes
3	EmptyList; Head	Legal	No
4	EmptyList; Next	Illegal	No
5	Append; EmptyList	Legal	No
6	Append; Append	Legal	No
7	Append; Head	Legal	Yes
8	Append; Next	Illegal	No
9	Head; EmptyList	Legal	No
10	Head; Append	Legal	No
11	Head; Head	Legal	No
12	Head; Next	Legal	Yes
13	Next; EmptyList	Legal	No
14	Next; Append	Legal	No
15	Next; Head	Legal	Yes
16	Next; Next	Legal	Yes

To test the data abstraction properly so that it can be used again, we should at least check the possible combinations of operations listed above on three representative list values: an empty list, a partially full list, and a full list.

Design, analysis, and testing provide a three-pronged approach to assuring confidence in the correctness and quality of the developing program. The design formalism assures that the program developed from the functional specification is a correct elaboration of that specification. The ability to perform a formal or informal analysis based on that design formalism offers the programmer and the reviewer the ability to check for consistency and correctness in a systematic way. Incremental testing of the program permits us to check details normally suppressed in proofs (e.g., restrictions on the sizes of objects), as well as the execution environment (e.g., the Pascal implementation corresponds to the formal definition).

IV. Experience with the Course

The course requires new ways of thinking, even for experienced computer science educators. The material is deep and quite different from the traditional "first course" in programming. However, the material is no harder for students than traditional introductory courses in other scientific disciplines such as mathematics or physics. We have a strong sense of satisfaction from teaching this course because it formalizes basic ideas of computer science and provides the student with a solid foundation in the principles of programming.

Because many students learn about computers before they come to the university, they enter with very different computer backgrounds. Those who know a lot about programming, even in Pascal, will not be able to skip this course. It is *not* a course in Pascal programming. The mix of backgrounds in programming and in mathematics causes student expectations and reactions to vary widely. Some are disappointed that they are not writing big realistic programs. They may think the material is too simple, until they are surprised by the real depth in the course. Others are initially overwhelmed because of their lack of background or intimidated by the programming knowledge of classmates. There will always be a few good students who have trouble simply because the course does not live up to their expectations. It is more important than usual to make it clear early what the course is about and what is expected. This motivation should be reinforced at regular intervals.

We distinguish between useful *tasks* and useful *skills*. Programming based on the program calculus and mathematical correctness is an unnatural activity that takes faith and practice to master. In learning to type, it is natural to look at the keys while typing. We know that we should learn to type in a systematic way (that is, without looking at the keys) because it is a better way to type in the long run, even though it is a terrible way to try to type on the first day. We learn not to do something that looks useful initially, but begin to learn useful skills instead. The improvement of mathematical approaches to programming over what comes naturally can be as dramatic as that of touch typing over hunt-and-peck.

Once they have mastered these ideas, programmers can use the Cleanroom software development method [10]. The key components to the method are a mathematically based design method, implementation without testing by developers, and statistically based functional testing performed by a third party. This method changes testing and debugging from program-development strategies to quality-assurance measures. A study of the development of an electronic message system by 15 three-person teams helped demonstrate the feasibility of this approach [13].

REFERENCES

[1] E. W. Dijkstra, "Notes on structured programming," in *Structured Programming*. New York: Academic, 1972, pp. 1-82.

[2] —, *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, J. T. Schwartz, Ed., 1967, pp. 19-32.

[4] J. D. Gannon, R. G. Hamlet, and H. D. Mills, "Theory of modules," *IEEE Trans. Software Eng.*, vol. SE-13, pp. 820-829, July 1987.

[5] C. A. R. Hoare, "An axiomatic approach to computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576-580, 583, Oct. 1969.

[6] —, "Proof of correctness of data abstraction," *Acta Inform.*, vol. 1, pp. 271-281, 1972.

[7] —, "Notes on data structuring," in *Structured Programming*. New York: Academic, 1972, pp. 83-174.

[8] H. D. Mills, "Mathematical foundations for structured programming," in *Software Productivity*. Boston, MA: Little, Brown and Co., 1982, pp. 115-178.

[9] H. D. Mills, V. R. Basili, J. D. Gannon, and R. G. Hamlet, *Principles of Computer Programming: A Mathematical Approach*. Boston, MA: Allyn and Bacon, 1986.

[10] H. D. Mills, M. Dyer, and R. Linger, "Cleanroom software engineering," *IEEE Software*, vol. 4, no. 5, pp. 19-25, Sept. 1987.

[11] P. Naur, "Proof of algorithms by general snapshots," *BIT*, vol. 6, pp. 310-316, 1966.

[12] D. L. Parnas, "A technique for software module specification with examples," *Commun. ACM*, vol. 15, no. 5, pp. 330-336, May 1972.

[13] R. W. Selby, V. R. Basili, and T. Baker, "Cleanroom software development: An empirical evaluation," *IEEE Trans. Software Eng.*, vol. SE-13, pp. 1027-1037, Sept. 1987.

[14] N. Wirth, "Program development by stepwise refinement," *Commun. ACM*, vol. 14, no. 4, pp. 221-227, Apr. 1971.

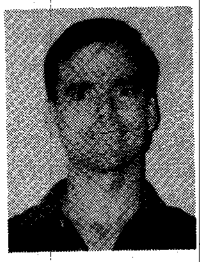
Harlan D. Mills (SM'82) received the Ph.D. degree from Iowa State University, Ames, in 1952.

He is the President and founder of the Information Systems Institute. He retired in 1987 as an IBM Fellow and as a Professor of Computer Science, University of Maryland, College Park. He continues to serve on the U.S. Air Force Scientific Advisory Board and as a part-time Professor of Computer and Information Sciences at the University of Florida, Gainesville. He was the principal architect for the curriculum of the IBM Software Engineering Institute, an IBM internal educational facility with a worldwide faculty of over 50. He served on the IBM Corporate Technical Committee and as Director of Software Engineering and Technology for the Federal Systems Division. He has also taught at Iowa State, Princeton, New York, and Johns Hopkins Universities. His industrial experience began with the General Electric Company in management consultation and operations research; he was a founder of the company Mathematica, serving as its first President.

Dr. Mills was the recipient of several distinguished awards, including honorary Fellow in Mathematics from Wesleyan University in 1962, ACPA Fellow in 1975, the DPMA Distinguished Information Sciences Award in 1985, and the Warnier Prize in 1987.

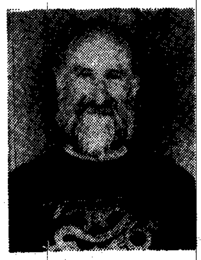
Victor R. Basili (M'83-SM'84) is Professor and Chairman of the Department of Computer Science at the University of Maryland, College Park. He was involved in the design and development of several software projects, including the SIMPL family of programming languges. He is currently measuring and evaluating software development in industrial and government settings and has consulted with many agencies and organizations, including IBM, GE, CSC, GTE, MCC, AT&T, Motorola, HP, NRL, NSWC, and NASA. He is one of the founders and principals in the Software Engineering Laboratory, a joint venture between NASA Goddard Space Flight Center, the University of Maryland and Computer Sciences Corporation, established in 1976. He has been working on the development of quantitative approaches for software management, engineering, and quality assurance by developing models and metrics for the software development process and product. He has authored over 90 papers. In 1982, he received the Outstanding Paper Award from the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING for his paper on the evaluation of methodologies.

Dr. Basili is currently the Editor-in-Chief of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and was Program Chairman for several conferences including the 6th International Conference on Software Engineering. He has served on the Editorial Board of the *Journal of Systems and Software*. He is a member of the Board of Governors of the IEEE Computer Society.



John D. Gannon received the A.B. degree in mathematical economics and the M.S. degree in applied mathematics from Brown University, Providence, RI, in 1970 and 1972, respectively, and the Ph.D. degree in computer science from the University of Toronto, Toronto, Ont., Canada, in 1975.

He is currently a Professor in the Department of Computer Science and the Institute for Advanced Computer Studies at the University of Maryland. His research centers on language and compiler design to increase the reliability of programs. Initially his work focused on the design of less error-prone programming languages. His interests in program proving and testing have led him to investigate formal specifications, test oracles, and test coverage metrics. He has also studied atomic remote procedure call as a primitive for distributed and fault-tolerant computing.



Richard G. Hamlet (M'81) is a Professor of Computer Science at Portland State University, Portland, OR. After a career in electrical engineering, engineering physics, and mathematics, he discovered computing, in the form of recursion theory, at the University of Washington, Seattle. He now concentrates on software engineering theory, especially testing theory.