[28] A. Birman, "On proving correctness of microprograms," *IBM J. Res. Develop.*, vol. 18, pr' 250-267, May 1974.
[29] C. A. R. Hoare, "Proof o a structured program: 'The sieve of Eratosthenes'," *Comput. J.* vol. 15, pp. 321-325, Nov. 1972.
[30] ——, "Proof of correctness of data representations," *Acta Informatica*, vol. 1, pp. 271-281, 1972.
[31] D. L. Parnas, "A technique for the specification of software modules with examples," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 330-336, May 1972.
[32] P. G. Neumann, "Toward a methodology for designing large systems and verifying their properties," Gesellschaft fur Informatik, Berlin, Germany, 1974.
[33] C. A. R. Hoare and N. Wirth, "An axiomatic definition of the programming language PASCAL," *Acta Informatica*, vol. 2, pp. 335-355, 1973.
[34] S. N. Zilles, "Algebraic specification of data types," Mass. Inst. Technol., Cambridge, Project MAC Progress Rep. 11, to be published.
[35] J. Donahue, J. D. Gannon, J. V. Guttag, and J. J. Horning, "Three approaches to reliable software: Language design, dyadic specification, complimentary semantics," Comput. Sci. Res. Group, Univ. of Toronto, Toronto, Ont., Canada, Tech. Rep. 45, Jan. 1975.
(36) C. Hewitt and 1. Greif, "Actor semantics of PLANNER-73," in *Proc. 2nd Ass. Comput. Mach. Symp. Principles oj Programming Languages*, Palo Alto, Calif., Jan. 20-22, 1975, pp. 67-77.

Barbara H. Liskov received the B.A. degree in mathematics from the University of California, Berkeley, and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, Calif.

From 1968 to 1972 she was associated with the Mitre Corporation, Bedford, Mass., where she participated in the design and implementation of the Venus Machine and the Venus Operating System. She is presently Assistant Professor of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, Cambridge. Her research interests include programming methodology and the design of languages and systems to support structured programming.

Stephen N. Zilles (S'71-M'73) was born in Toledo, Ohio, on July 1, 1941. He received the S.B. degree in economics and polititaI science in 1963 and in mathematics in 1967 from the Massachusetts Institute of Technology, Cambridge. In 1970, he received the S.M. and E.E. degrees in electrical engineering, also from the Massachusetts Institute of Technology.

From 1963 to 1968 he was employed by the IBM Corporation, San Jose, Calif., where he worked on the compilation of PL/I, an interactive system for building large programs and Formac, a system for algebraic manipulation. While completing the M.S. and E.E. degrees at the Massachusetts Institute of Technology he was a Teaching Assistant, and a Research Assistant in Programming Linguistics at MIT Project MAC. Upon returning to IBM in 1970 he was Advanced Programming Manager responsible for implementing a prototype language interpreter. Since 1971 he has been active in the design and evaluation of programming and command languages at IBM. His current interests are in the semantics and design of programming languages with particular emphasis on the description and representation of data types.

Mr. Zilles is the Secretary-Treasurer of the Association for Computing Machinery SIGPLAN and a member of the Association for Computing Machinery Board on Special Interest Groups and Committees. He is a member of Sigma Xi and the Association for Computing Machinery.

On Preventing Programming Languages from Interfering with Programming

W. M. McKEEMAN, MEMBER, IEEE

***Abstract*—Wirth** has proposed a method of "stepwise refinement" for writing computer programs. This paper proposes that the steps be **expressed** as proofs. A program for the eight-queens problem is developed, and the proof method is applied across two of the steps of the development. The strengths and weaknesses of the method, and its implications for the programming process and programming **language** design are **discussed.**

Index ***Terms*—Adaptability,** correctness, portability, software engineering, structured programming.

Manuscript received November 1, 1974; revised January 20, 1975. This work was supported in part by the Computer Systems Research Group, University of Toronto, Toronto, Ont., Canada.

The author is with the Department of Information Sciences, University of California. Santa Cruz. Calif. 95064.

1. INTRODUCTION

THE PROBLEM of getting programs written reliably and economically is of considerable importance. Wirth [12J proposed a method of "stepwise refinement" which starts from an arbitrarily chosen symbolic notation and goes through a number of steps, each getting closer to the chosen programming language. Naur [9J presented a personal account of applying a similar method to the problem of eight queens. His report exposed an uncomfortably believable series of false starts and uncertain steps leading to a successful solution. Knuth [6J carried the concept further by pointing out that even after a programming language version is achieved, the program-

mer can continue to refine his program to improve its efficiency. In all of the above, the authors hand translated the program from one intermediate form to another.

One view of the programming process is that there is a long sequence of intermediate forms of the program. It starts in some form in the mind of the programmer. The first (and perhaps the hardest) step in the process is to get it into symbolic form, usually in a natural language. The programmer may then apply the method of stepwise refinement until an efficient programming language version is achieved. The rest of the process is mechanized, leading through compilation, and finally to some form in the state of the physical components of a computer. In this paper, we are concerned with the reliability and economy of the programming process prior to the first mechanical translation step.

We propose to call each intermediate form of a program a "form" and the process of getting from one form to another a "step." Forms will correspond to theorems and steps to proofs. A complicated program may show enough structure to allow the formulation of subprograms with independent sequences of forms and steps, merging to give the whole. The task of programming can be identified with the task of creating a complete sequence of forms connected by informal steps. The task of formalizing the steps can be identified with the task of program proving.

Wirth points out that even a short program may be best derived through a long sequence of steps, corresponding to the many design decisions actually being made. While it is only the final form that is needed to communicate with the computer, it is implicit in the method that the complete, carefully refined sequence is the proper form for communication between programmers.

In the process of proving the steps, the programmer reveals (to himself) the assumptions upon which the steps depend. The primary motivation for making the assumptions explicit is that occasionally they turn out to be false, thereby calling into question the validity of the program.

Hoare [3J takes a similar approach in his proof of the program FIND. He analyzes the program "top-down," each successive stage embodying more detail than the previous one. He then proves the correctness of the program at each stage (rather than its equivalence to the previous stage). His proof statements survive in the final program as assertions relating the states of the program variables. His proof method has some advantages over the one proposed here; his proof is complete and convincing (as compared to the partial result of this paper); his notation is consistent throughout, starting and ending in a programming language.

On the other hand, the method proposed here is based on the assumption that it may be advantageous to delay the appearance of programming notation until late in the refinement process. It allows the separation of those aspects of "correctness" that have to do with the algorithm and those that have to do with the programming language notation.

Section II of this paper consists of a sequence of refinements for the eight-queens problem. In Section III, certain steps are verified with formal proofs.

II. SOLUTION OF THE EIGHT-QUEENS PROBLEM

Form 1 is the problem statement; it is partial and informal, but precise.

> Find all ways eight queens can be placed on a chessboard so that no pair is mutually threatening.
>
> Form 1

The next form is an equivalent statement indicating the formal approach and naming two major constructs, BOARD and P, to be used in later analysis. P is, in fact, the solution.

> BOARD is the set of all squares on the chessboard;P is the set of all subsets of BOARD such that
> 1) each contains eight squares, and
> 2) queens on each pair of squares are not mutually threatening.
>
> Form 2

We can immediately translate into a formal set-theoretic notation. Both the set BOARD and the function THREATEN are still only informally defined.

$$P \quad \{X \mid X \subseteq \text{BOARD} \wedge \text{SIZE}(X) = 8 \wedge (x \in X \wedge y \in X \Rightarrow \text{ITHREATEN}(X,y))\}$$

Form 3

We remark that the function THREATEN must have the properties

$$\text{THREATEN}(X,y) = \text{THREATEN}(y,X)$$
$$\text{THREATEN}(X,X) = \text{false}.$$

We now proceed, in Form 4, to sequentialize the implied computation by constructing BOARD to be the union of its columns (as Dijkstra suggested to Wirth [12, p. 275J):

$$\text{BOARD} \equiv C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5 \cup C_6 \cup C_7 \cup C_8.$$

We also define a function $T(X)$ giving all the squares threatened by all the queens on squares in X. The test implied in Form 4, $y \notin T(X)$, is apparently less efficient than the equivalent test, $y \notin T(X) \cap C_k$, but it leads to an improved formalization in the subsequent stage.

$$P_0 = \{\{\}\};$$
$$P_k = \{X \cup \{y\} \mid X \in P_{k-1} \wedge Y \in C_k \wedge Y \notin T(X)\};$$
$$T(X) = \{y \mid x \in X \wedge y \in \text{BOARD} \wedge \text{THREATEN}(X,y)\}$$

Form 4

In Form 5 we turn the problem inside out. Instead of collecting sets of queen-occupied squares, we collect sets of queen-threatened squares. The effect is to move the function T from a position where it operates on the large set X to a position where it operates on a set with a single element. The result is that it has only 64 possible values and can be tabulated prior to the main computation.

$Q_0 = \{\{\}\};$
$Q_k = \{Y \cup T(\{x\}) \mid Y \in Q_{k-1} \wedge x \in C_k \wedge x \notin Y\};$
$T(\{x\}) = \{y \mid y \in \text{BOARD} \wedge \text{THREATEN}(x,y)\}.$

Form 5

We now proceed to concentrate on changing from set-theoretic notation towards a programming language. We choose Algol-W [11] and will use its constructs in subsequent forms of the program. When restrictions of Algol-W interfere, we shall feel free to make obvious extensions to the language (to be eliminated by the final form, of course).

Small sets can be mapped directly onto type BITS in Algol-W. A "1" in position i of a variable of type BITS signifies that the ith element is in the corresponding set. The set operations of "union" and "intersection" map onto the primitive machine operations of OR and AND.

Since BOARD has 64 elements, the corresponding programming language variable representing a subset of BOARD must have 64 bits. In order to translate between the coordinate location of a square and the BITS representation of a square, we can initialize

BITS(64) ARRAY U(1::8,1::8)

so that the variable $U(i,j)$ has exactly one "1" in the position corresponding to the square i,j of the chessboard.

If, furthermore, we initialize

BITS(64) ARRAY T(1::8,1::8)

so that each variable $T(i,j)$ has all bits "1" corresponding to the squares threatened by a queen on square i,j, the computation in Form 6 is equivalent to that of Form 5. (The symbol "#" starts a constant bit string in Algol-W; #0 signifies a string with all zeros.)

```
BITS(64) ARRAY T, U(1::8,1::8);
Q0 = {{}};
Qk = {Y OR T(i,k) | Y ∈ Qk-1 ∧ 1 ≤ i ≤ 8 ∧
                    (Y AND U(i,k)) = #0}.
```

Form 6

The algorithm is now directly implementable, but the iteration on k forces us to simulate the accumulation of the sets Q_k for which there is no ready linguistic construct in Algol-W. The iteration can be turned into a recursion, yielding the procedure in Form 7.

```
PROCEDURE Q(BITS(64) VALUE Y; INTEGER VALUE k);
  IF k = 9 THEN PRINTBOARD(,Y)
  ELSE FOR i := 1 UNTIL 8 DO
    IF (Y AND U(i,k)) = #0
    THEN Q(Y OR T(i,k), k + 1);
```

Form 7

The problem can now be solved by a procedure call of the form

Q(#0,1).

Once having started Q off, it will call itself repeatedly. The set of values for the first parameter over all the calls is exactly the set of values in all the sets Q_k. When the second parameter has value 8, the recursive calls have parameters corresponding to the elements of Qs (and second parameter of value 9). Thus it is the elements of Qs that are printed.

We must provide for the initialization of the arrays T and U. Since the only requirement for U is that unique bit positions correspond to unique pairs i,j, the loop in Form 8 is sufficient.

```
BITS(64) V;
BITS(64) ARRAY U(1::8,1::8);
V := #8000000000000000;
FOR i := 1 UNTIL 8 DO FOR j := 1 UNTIL 8 DO
BEGIN
  U(i,j) := V; V := V SHR 1;
END;
```

Form 8

The array T poses a harder problem. We must finally explore the function THREATEN in detail, starting in Form 9.

A queen on square x threatens another queen on square y if

1) x and y are in the same row;
2) x and y are in the same column;
3) x and y are on the same positively sloping diagonal;
4) x and y are on the same negatively sloping diagonal.

Form 9

We can proceed to a definition of the function $T(\{x\})$ from Form 5. The argument $\{x\}$ can be represented by its coordinate pair i,j. The value of the function is the union of the row, column, and two diagonals containing x (less x itself).

```
T(i,j) = (Row(i,j) ∪ COL(i,j) ∪ PD(i,j) ∪ ND(i,j»
                                        - {(i,j)};
Row(i,k) = {(i,j) | 1 ≤ j ≤ 8};
cOL(h,j) = {(i,j) | 1 ≤ i ≤ 8};
PD(h,k) = {(i,j) | h - k = i - j ∧ 1 ≤ i ≤ 8
                                        ∧ 1 ≤ j ≤ 8};
ND(h,k) = {(i,j) | h + k = i + j ∧ 1 ≤ i ≤ 8
                                        ∧ 1 ≤ j ≤ 8}.
```

Form 10

We could proceed to implement ROW, COL, PD, and ND as functiqns; it is more efficient to define arrays of the same name and precompute all their values. We can drop one coordinate from each of Rowand COL, and need only have values of PD and ND for constant sum and difference of coordinates. The first subtask is to initialize arrays ROW and COL with bits corresponding to threatened rows and columns.

```
BITS(64) ARRAY ROW, cOL(I: :8);
FOR i := 1 UNTIL 8 DO
BEGIN
  ROW(i) := COL(i) := #0;
  FOR j := 1 UNTIL 8 DO
  BEGIN
    ROW(i) := Row(i) OR U(i,j);
    COL(i) := COL(i) OR U(j,i);
  END;
END;
```

Form 11

The second subtask is the initialization of the arrays PD and ND. Following the definitions in Form 10, we get the program fragment in Form 12.

```
BITS(64) ARRAY PD(-7: :7);
BITS(64) ARRAY ND(2: :16);
FOR i := -7 UNTIL 7 DO PD(i) := #0;
FOR i := 2 UNTIL 16 DO ND(i) := #0;
FOR i := 1 UNTIL 8 DO FORj := 1 UNTIL 8 DO
BEGIN
  PD(i - j) := PD(i - j) OR U(i,j);
  ND(i + j) := ND(i + j) OR U(i,J);
END;
```

Form 12

Finally, they are combined in the array *T*.

```
BITS(64) ARRAY T(I: :8,1: :8);
FOR i := 1 UNTIL 8 DO FORj := 1 UNTIL 8 DO
BEGIN
  T(i,j) := Row(i) OR COL(j) OR PD(i - j) OR
                                        ND(i + j);
  COMMENT The occupied square is not threatened;
  T(i,j)    T(i,j) AND iU(i,j);
END;
```

Form 13

Nothing else remains to be done, except the combination of the pieces into the program in Form 14, and the programming of double precision type BITS since Algol-W does not provide for a type LONG BITS.

```
BEGIN COMMENT McKeeman's program for the eight
  queens;
  BITS ARRAY TI, T2, UI, U2(I: :8,1: :8);
  PROCEDURE INITIALIZE;
  BEGIN
    BITS VI, V2;
    BITS ARRAY RowI, Row2, COLI, cOL2(I: :8);
    BITS ARRAY PDI, PD2(-7: :7);
    BITS ARRAY NDI, ND2(2: :16);
    COMMENT Prepare mapping between ordered pairs
    and bit positions;
    VI := #80000000;    V2:= #0;
    FOR i := 1 UNTIL 8 DO FORj := 1 UNTIL 8 DO
    BEGIN
      UI(i,j) := VI;    VI := VI SHR 1;
      U2(i,j) := V2;    V2 := V2 SHR 1;
      IF (VI = #0) AND (V2 = #0) THEN V2 :=
      #80000000;
    END;
    COMMENT Set patterns for threatened rows and
    columns;
    FOR i := 1 UNTIL 8 DO
    BEGIN
      ROW1(i) := ROW2(i) := #0;
      COLI(i) := COL2(i) := #0;
      FORj := 1 UNTIL 8 DO
      BEGIN
        ROW1(i) := ROW1(i) OR UI(i,J);
        ROW2(i) := ROW2(i) OR U2(i,J);
        COL1(i) := COL1(i) OR U1(j,i);
        COL2(i) := COL2(i) OR U2(j,i);
      END;
    END;
    COMMENT Set patterns for threatened diagonals;
    FOR i := -7 UNTIL 7 DO PD1(i) := PD2(i) := #0;
    FORi:= 2 UNTIL 16 DO ND1(i) := ND2(i) := #0;
    FOR i := 1 UNTIL 8 DO FORj := 1 UNTIL 8 DO
```

```
    BEGIN
      PD1(i - j) := PD1(i - j) OR UI(i,j);
      PD2(i - j) := PD2(i - j) OR U2(i,j);
      ND1(i + j) := ND1(i + j) OR UI(i,j);
      ND2(i + j) := ND2(i + j) OR U2(i,j);
    END;
    COMMENT Combine threatened rows, columns and
    diagonals;
    FOR i := 1 UNTIL 8 DO FOR j := 1 UNTIL 8 DO
    BEGIN
      TI(i,j) := Row1(i) OR COL1(j) OR
                 PD1(i - j) OR ND1(i + j);
      T2(i,j) := Row2(i) OR COL2(j) OR
                 PD2(i - j) OR ND2(i + j);
      COMMENT Occupied square is not threatened;
      TI(i,j) := TI(i,j) AND jUI(i,j);
      T2(i,j) := T2(i,j) AND jU2(i,j);
    END;
  END INITIALIZE:
  PROCEDURE PRINTBOARD(BITS VALUE Yi, Y2);
    WRITE(Y1,Y2);
  PROCEDURE Q(BITS VALUE YI,Y2; INTEGER
    VALUE k);
    IF k = 9 THEN PRINTBOARD (¬Y1, ¬Y2)
    ELSE FOR i := 1 UNTIL 8 DO
      IF «YI AND U1(i,k) = #0) AND «Y2 AND
        U2(i,k» = #0)
      THEN Q(YI OR TI(i,k), Y2 OR T2(i,k), k + 1);
  INITIALIZE;
  Q(#0, #0, 1);
END.
```

Form 14

There are undoubtedly more efficient solutions. to the eight-queens problem in Algol-*W*. Having achieved a solution in Form **14,** we might continue by simulating recursion with. a stack, and so on. Were that an important issue (which it was not in this case; the program ran in about 13 s on our IBM S/360 model 40), we would feel obligated to continue. Furthermore, the assumptions that lead from step to step may not have been introduced in the optimal order from the viewpoint of generality. How much of the analysis can be used to solve the eight-rook problem, or the similar problem for bishops or knights?

III. PROOF OF THE EIGHT-QUEENS SOLUTION

There is a *significant property,* expressed as a predicate on the data structures of the two forms, which must hold across a step. The property may have many facets, some related to correctness, some related to efficiency, Some related to generality. Proofs are given here for two of the steps of the preceding section. For Step 3–4, the significant property is that the set P of Form 3 is equivalent to set P_8 of Form 4. For Step 4-5, the significant property is that sets in P_a of Form 4 are the complements of sets in Q_a of Form 5.

Step 3–4: Show $P \equiv P_a$.

We first define partial boards:

$$\text{BOARDk} = \bigcup_{1 \le i \le k} C_i$$

and show instead that

1) $P_k \subseteq \{X \mid X \subseteq \text{BOARDk} \wedge \text{SIZE}(X) = k \wedge (x \in X \wedge y \in X \Rightarrow \text{jTHREATEN}(X,y)\}$

which for $k = 8$ gives $P_a \subseteq P$, and show also that

2) $\{X \cap \text{BOARDk} \mid X \in PI\} \subseteq P_k$

which for $k = 8$ gives inclusion the other direction, $P \subseteq P_a$.

We establish proposition 1) by induction. Since BOARD0 is null, we have

$$\{X \mid X \subseteq \text{BOARD0} \wedge \text{SIZE}(X) = 0 \wedge (x \in X \wedge y \in X \Rightarrow \neg \text{THREATEN}(x,y))\} = \{\{\}\}$$

establishing the inclusion for $k = 0$. Now assume the inclusion holds for $k - 1$. If we can show the three propositioilS

3) $X \in P_k \Rightarrow X \subseteq \text{BOARDk}$,

4) $X \in P_k \Rightarrow \text{SIZE}(X) = k$, and

5) $X \in P_k \Rightarrow (x \in X \wedge y \in X \Rightarrow \text{jTHREATEN}(X,y))$

we have proposition 1) for all k. Proposition 3) is trivial. For proposition 4) we note from the definition of P_k that either we add a single element to each member of P_{k-1} to get P_k or P_k is null. In either case, proposition 4) holds.

Now suppose $X \in P_k$. Then by definition, there exists a $Y \in P_{k-1}$ and $y \in C_k$ such that $X \equiv Y \cup \{y\}$' and

6) $x \in Y \Rightarrow \text{jTHREATEN}(X,y)$.

Now

$$X \times X \equiv (Y \times Y) \cup (Y \times \{y\}) \cup (\{y\} \times Y) \cup (\{y\} \times \{y\}).$$

Each of the four subsets of arguments to THREATEN now can be shown to have the value false. For the first, by the induction hypothesis,

$$x \in Y \wedge y \in Y \Rightarrow \text{jTHREATEN}(X,y).$$

For the second and third, proposition 6) and

$$\text{THREATEN}(X,y) = \text{THREATEN}(y,X)$$

is sufficient. For the fourth, we have

$$\text{THREATEN}(y,y) = \text{false};$$

hence we get proposition 5). Propositions 3), 4), and 5) together give 1), giving, for $k = 8$,

$$P_a \subseteq P.$$

To get inclusion the other way, we may show proposition 2). First, however, we need to show that

$$X \in P \Rightarrow \text{SIZE}(X \cap Ck) = 1 \quad \text{for } 1 \leq k \leq 8.$$

Because the C_k are disjoint, $X \in P$ implies

$$\text{SIZE}(X) = \sum_{k=1,S} \text{SIZE}(X \cap Ck).$$

But $\text{SIZE}(X \cap Ck) \leq 1$ since

$$x \in C_k \wedge Y \in C_k \wedge x \neq Y \Rightarrow \text{THREATEN}(x,y).$$

Therefore, since SIZE(X) = 8,

$$\text{SIZE}(X \cap C_k) = 1 \quad \text{for } 1 \leq k \leq 8.$$

Notice that we had to formalize a third property of THREATEN arid use the disjointedness of the *Ck*.

Returning to the main argument, let

$$Y \in \{X \cap \text{BOARD}_k \mid X \in Pl.$$

Then there is an $X \in P$ such that

$$Y \equiv X \cap (\text{BOARD}_{k-1} \cup C_k)$$
$$\equiv (X \cap \text{BOARDk\_l}) \cup (X \cap Ck).$$

Because $\text{SIZE}(X \cap C_k) = 1$, there exists $a y \in C_k$ such that

$$Y \equiv (X \cap \text{BOARDk\_l}) \cup \{y\}.$$

By the definition of P,

$x \in (X \cap \text{BOARDk}) \wedge y \in (X \cap \text{BOARDk}) \Rightarrow \text{ITHREATEN}(x,y)$

so that (recalling $y \in X \cap C_k$)

$$x \in (X \cap \text{BOARDk\_l}) \Rightarrow \text{ITHREATEN}(x,y),$$

completing the satisfaction of the definition of P_k; hence

$$Y \in P_k$$

and therefore

$$\{X \cap \text{BOARD}_k \mid X \in P\} \subseteq P_k$$

and,finally, fork = 8,

$$P \subseteq Ps,$$

giving the required equivalence

$$P \equiv P_S.$$

The proof was surprisingly difficult, revealing that the step involved several assumptions about the nature of chessboards and the rules for threatening chesspieces.

Step 4-5 is a nontrivial reformulation of the problem, and can be expected to involve new assumptions.

Step 4-5: Show $X \in P_S$ if and only if BOARD - X E Qs. We first note that the definitions of T in Forms 5 and 6 are equivalent. Then we establish, by induction,

$$Q_k \equiv \{T(X) \mid X \in P_k\}$$

and, finally, recalling $P \equiv P_S$ froin Step 3-4,

$$X \in P \Rightarrow \text{BOARD} - X \equiv T(X).$$

Proceeding, $Po \equiv \{\{\}\}$, $T(\{\}) \equiv \{\}$, $Qo \equiv \{\{\}\}$; hence $Qo \equiv \{T(X) \mid X \in Po\}$. Assume the induction formula for k - 1. Then for each $X \cup \{y\} \in P_k$, $X \in P_{k-1}$, and therefore $T(X) \in Qk\text{-}l$. By expanding the definition of T (Form 4); we get

$$T(X \cup \{y\}) \equiv T(X) \cup T(\{y\});$$

hence

$$Q_k \equiv \{T(X) \mid X \in P_k\}.$$

Now assume $X \in P$. We have

$$X \cap T(X) \equiv \{\}$$

for otherwise there is a y in both and an x in X such that both

$$\text{THREATEN}(X,y)$$

and

$$\text{ITHREATEN}(X,y)$$

hold.

Furthermore,

$$X \cup T(X) \equiv \text{BOARD}$$

for otherwise there is a y in neither and an x in X such that neither

$$\text{THREATEN}(x,y)$$

nor

$$\text{ITHREATEN}(x,y)$$

hold;

Hence

$$\text{BOARD} - X \equiv T(X).$$

IV. CONCLUSIONS

We have outlined three views of what a program ought to be. The final form of development is suitable for communIcation to a computer; a carefully documented sequence of forms is suitable for communication between programmers; a proven sequence of forms is suitable for communication to posterity. There are some important implications for the practice of programIning and the art of programIning language design.

Knuth, to ameliorate the evangelism surrounding the eliInination of the GOTO, recalls the unsuccessful attempt of the "Dutch School" of mathematics to impose strictly constructionist methods on all of matheinatics [5]. There is a closer parallel: the application of assertions to programs [2]. The problem with assertions, like that with the constructionist methods, is that they deny "almost all" convincing proofs that we programxners might come up with to validate our programs. That is not to say we should not use them where we can. And it is certainly not meant to imply that we may not impose them on our students, much as we now deny students the GOTO. But we must realize that assertions are too constraining to become the only acceptable method for proving programs correct.

Since the sequence of forms expressing the solution to a particular programming problem is not unique, we may consider the problem of choosing the best among alternatives. This, in turn, implies a global measure of efficiency including not only space and time during execution, but also in the information processing involved in the creation, proving, and maintenance of the program, and in the modifications of it for related problems and environments [10]. If only a small part of the whole sequence of program forms need be changed to respond to a demand, the global measure of efficiency is obviously enhanced. This would imply that, from among all the assumptions needed to prove the various steps, those that depend upon volatile data (such as the cost or speed of hardware) should be pushed as far as possible towards the end of the sequence. In particular, optimizations should be last. It is here that the additional structure of this method can be exploited.

Criticism of and apologies for clever programming have lately been appearing in print. This seems strange considering it is the really clever solutions to hard problems that have been traditionally honored by the scientific community. We cannot mean that clever new algorithms are to be denied. And we cannot mean that clever new ways to optimize programs are to be denied. The problem has arisen when we have done both simultaneously. And that has occurred largely because both problem solution and optimization have been expressed in the same form. The method of stepwise refinement provides a medium for keeping them separate, making them understandable, and thereby returning cleverness to social acceptability.

I now believe that it was a fundamental error to feel that we could think in a programming language [8]. I once stated "that the major responsibility for computer language design should rest with the language user," and proceeded to attempt to bring compiler writing to a state where a serious programmer could consider implementing a compiler as a part of the problem solving process [7]. The results fell short of the promise. A major reason was that *implementability* was as contraining as *constructionist methods.* Programmers can profitably deal with partially defined languages, infinite objects, and all manner of heady reasoning. Computers probably can also. But, for the time being, it is easier for programmers, and thus "global efficiency" says that they, not the machines, should do it. It is freedom with a price; the programmer is free to invent any notation that he is willing to translate into an implemented programming language, and for which the translation steps can be verified by convincing formal proofs.

There is little advantage to be gained in the process outlined here by having a target programming language which is "close" to mathematical notation. The mental leap from sets to bits is not aided by calling bits "sets," except for the programmer, to whom the thought has never occurred. Furthermore, the type "SET" in PASCAL, for example, could be misleading to the intuition since the obvious generalizations to sets of 61 elements, or sets of sets, and so on, are not implemented [12]. It seems better not to gloss over the actual limitations of the computer with a respectable but not entirely faithful mathematical pattern unless the differences are so remote that the programmer rarely blunders into them.

That is not to say that we ought not to restrict the use of the mechanisms that are available through appropriate language structures (such as EXIT instead of GOTO). What we need are programming languages that are easy targets for stepwise refinement, whatever that entails.

A final comment is elicited by Knuth's reference to the "Shanley design criterion" [6]. When two "neighboring" functions can be combined with a gain in efficiency, they should be. In Form 4 of the eight-queens sequence, the computation of $T(X)$ is "near" the computation of $X \cup \{y\}$. In Form 5, their "nearness" is exploited to eliminate a whole class of repetitive computations. Such phenomena are very common in programming. As in the case of other engineering disciplines, combination is likely to be tricky, requiring a greater depth of understanding to bring it off. Stepwise refinement provides the opportunity to develop the functions separately and then document and prove their combination.

On the other hand, we are told that programs (meaning the final form) should be readable [4]. If it seems like bad policy to expect an engineer to understand a Saturn rocket from cutting one up and watching another one fly, it would also seem like bad policy to expect a programmer to understand a carefully optimized program without recourse to the "plans" embodied in the sequence of forms of development. The requirement is much stronger: the entire sequence of forms, including the last, must be "readable.',

In summary, let me express some final thoughts that seem to bring these ideas into focus for me. First, programmers need at least the power available to modern mathematicians. Second, the ideal of "a" programming language, even for the solution of a single problem, contradicts that need. Third, that "intermediate languages in the region of human translation" are useful tools for the creative programmer. Fourth, that a well thought-out, correct sequence of forms "converging" to an efficient program in an implemented programming language is the preferable unit of communication between programmers. And, finally, when dealing with formal systems, there is no substitute for formal proofs.

ACKNOWLEDGMENT

The author is indebted to the participants of the 1974 Workshop on the Achievement of Reliable Software, Toronto, Ont., Canada, whose spirited discussions were the starting point for this paper. The initial draft was written with a great deal of help from Prof. Horning and his students J. Guttag and J. Gannon of the Computer Systems Research Group, University of Toronto. Thanks are also due Prof. C. A. R. Hoare and the referees for many helpful suggestions.

REFERENCES

[1] E. 'V. Dijkstra, "Notes on structured programming," in *Structured Programming,* O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Ed. New York: Academic, 1972.

[2] Q. A. R. Hoare, "An axiomatic basis for computer programming." *Commun.* Ass. *Comput. Mach.,* vol. 12, pp. 576-581, Oct. 1969.

[3] ——, "Proof of a program: FIND," *Commun.* Ass. *Comput. Mach.,* vol. 14, pp. 39-45, Jan. 1971.

[4] B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style.* New York: McGraw-Hill, 1974.

[5] S. C. Kleene, *Introduction to Metamathematics.* New York: Van Nostrand Reinhold, 1952.

[6J D. E. Knuth, "Structured programming with GOTO statements," *Comput. Surveys,* pp. 261-301, Dec. 1974.

[7] 'V. M. McKeeman, "An approach to computer language design," Ph.D. dissertation, Stanford Univ., Stanford, Calif., Apr. 1966.

[8] ——, "Programming language design," ch. 5.C in *An Advanced Course in Compiler Construction.* New York: Springer-Verlag, 1974.

[9J P. Naur, "An experiment in program development," *BIT,* vol. 12, pp. 347–365, 1972.

[10] P. C. Poole and 'V. M. Waite, "Portability and adaptability," in *An Advanced Course on Software Engineering,* vol. 81. New York: Springer-Verlag, 1973.

[11] N. Wirth and C. A. R. Hoare, "A contribution to the development of Algol," *Commun.* Ass. *Comput. Mach.,* vol. 9, June 1966.

[12] N. Wirth, "Program development by stepwise refinement," *Commun.* Ass. *Comput. Mach.,* vol. 14, pp. 221-227, Apr. 1971.

[13] ——, "The programming language PASCAL," *Acta Inform.,* vol. 1, pp. 35-63, 1971.

W. M. McKeeman (M'71) received the B.A. degree in mathematics from the University of California, Berkeley, the M.A. degree in mathematics from the George Washington University, Washington, D. C., and the Ph.D. degree in computer science from Stanford University, Stanford, Calif.

He has held teaching positions at the U. S. Naval Academy, Annapolis, Md., the University of Toronto, Toronto, Ont., Canada, and Stanford University. He is currently Chairman of Information Sciences at the University of California, Santa Cruz, TechnIcal Coordinator of the Computer Science Institute of the University Extension, and Executive Vice President of Data Base Management SysteIllS, Inc. His major interests are in the design, use, and evaluation of computer systeIllS.

Towards a Programming Apprentice

CARL E. HEWITT AND BRIAN SMITH

Abstract-The PLANNER Project is constructing a *programming apprentice* to assist in knowledge based programming. We would like to provide an environment which has substantial knowledge of the semantic domain for which the programs are being written, and knowledge of the purposes that the programs are supposed to satisfy. Further, we would like to make it easy for the programmer to communicate the knowledge about the program to the apprentice. The apprentice is to aid expertprogrammers in the following kinds of activities:

1) establishing and maintaining consistency of specifications;
2) validating that modules meet their specifications;
3) answering questions about dependencies between modules;
4) analyzing implications of perturbations in modules; and
5) analyzing implications of perturbations in specifications.

We use *contracts* (procedural specifications) to express what is supposed to be accomplished as opposed to how it is supposed to be done. The idea is that at least two procedures should be written for each module in a system. One procedure implements a method for accomplishing a desired transformation and the other can check that the transformation has in fact been accomplished. The programming apprentice is designed for interactive use by expert programmers in the meta-evaluation of implementations in the context of their contracts and background knowledge. Meta-evaluation produces a *justification* which makes explicit exactly how the module depends on the contracts of other modules and on the background knowledge. The justification is used in answering questions on the behavioral dependencies between modules and in analyzing the implications of perturbations in specifications and/or implementation.

Manuscript received December 15, 1974; revised February 3, 1975. This research was sponsored by the Massachusetts Institute of Technology Artificial Intelligence Laboratory and Project MAC under a contract from the Office of Naval Research. A preliminary version of this paper was presented at the Artificial Intelligence and Simulation of Behavior Conference, Sussex, England, July 1974.

The authors are with the Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass. 02139.

INTRODUCTION

A TENET of the apprentice project is that programming is *a multilevel activity:* as well as writing code, programmers communicate in terms of comments and models. Programmers develop and are given specifications for a model of what they want to compute. Code is an embodiment of that model. Comments are an attempt to elucidate the relationship between the model and the code. Our goal is to elucidate and formalize some of these interactions. The first level of description we have attacked is the level of abstract descriptions of *what* programs do, rather than *how* they do it. The contracts and intentions discussed in this paper are an attempt to embody this kind of knowledge in a formal and yet intuitive and useful way. A process known as *meta-evaluation* [25J is presented which can justify that a program fulfills its contract. Further research is being carried out into the role of models,