Copyright © 1981 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 60, No. 3, March 1981
Printed in U.S.A.

Program Development by Stepwise Refinement and Related Topics

By N. GEHANI

(Manuscript received September 22, 1980)

Computer program development by stepwise refinement has been advocated by many people. We take another look at stepwise refinement in light of recent developments in programming languages and programming methodology such as abstract data types, correctness proofs and formal specifications, parallel programs and multiversion programs. We offer suggestions for the refinement process and discuss program maintainability.

I. INTRODUCTION

The correct design of nontrivial programs and systems of programs is an intellectually challenging and difficult task. Often programs are designed with very little time spent on the design itself, the effort being concentrated on coding. This could be due to management's desire to see something working as soon as possible to be assured that work is progressing, or it could be due to the programmer's desire to "attack the problem right away."

Not only is there no emphasis on design, the approach to it is also not systematic or disciplined. This results in programs that do not meet specifications in terms of correct output and performance requirements.

What we want is a programming methodology that puts some discipline and structure in the design process without stifling creativity. A programming methodology should:

(*i*) Help us master the complexity of the problem being solved and give us some guidelines on how to formulate the problem solution.

(*ii*) Provide us with a written record of the design process. The design can then be read by others, and the design decisions can be appreciated or constructively criticized.

(*iii*) Result in programs that are understandable.

(*iv*) Lead to programs whose correctness can be verified by proofs. Since proofs are difficult, the methodology should allow for a systematic approach to program testing.

(*v*) Be generally applicable and not restricted to a class of problems.

(*vi*) Allow for the production of efficient programs.

(*vii*) Allow for the production of programs that can be modified systematically.

In this tutorial we discuss a programming methodology called stepwise refinement and informally show that it satisfies these criteria.

II. STEPWISE REFINEMENT

Stepwise refinement is a top-down design approach to program development (first advocated by Wirth[4]). Wirth really gave a systematic formulation and description of what many programmers were previously doing intuitively. According to Brooks,[2] stepwise refinement is the most important new programming formalization of the decade. Stepwise refinement is applicable not only to program design, but also to the design of complex systems.

In a top-down approach, the problem to be solved is decomposed or refined into subproblems which are then solved. The decomposition or refinement should be such that:[3,4]

(*i*) The subproblems should be solvable.

(*ii*) A subproblem should be solvable with as little impact on the other subproblems as possible.

(*iii*) The solution of each subproblem should involve less effort than the original problem.

(*iv*) Once the subproblems are solved, the solution of the problem should not require much additional effort.

This process is repeated on the subproblems; of course, if the solution of a problem is obvious or trivial, then this decomposition is not necessary.

If P_0 is the initial problem formulation/solution, then the final problem formulation/solution P_n (an executable program) is arrived at after a series of gradual "refinement" steps,

$$P_0 \Rightarrow P_1 \Rightarrow P_2 \Rightarrow \cdots \Rightarrow P_n.$$

The refinement P_{i+1} of P_i is produced by supplying more details for the problem formulation/solution P_i. The refinements $P_0, \cdots, P_n$ represent different levels of abstraction. P_0 may be said to give the most abstract view of the problem solution P_n, while P_n represents a detailed version of the solution for P_0.

As an example of abstraction levels, consider a program that automates the record-keeping of an insurance company. At the highest level of abstraction, the program deals with the insurance company as an entity. At succeeding lower levels of abstraction, the program deals with

- different insurance categories (auto, home, life, etc.)
- groups of policies in the above categories
- individual policies in the above groups
- details of individual policies

Each refinement P_i consists of a sequence of instructions and data descriptions P_{ij},

$$\begin{array}{c} P_{i1} \\ \vdots \\ P_{in_i} \end{array}$$

In each refinement step, we provide more details on how each P_i is to be implemented. The refinement process stops when we reach a stage
(*i*) where all the instructions can be executed on a computer, or
(*ii*) where instructions can be easily translated to computer executable instructions.

Pictorially, the refinement process may be depicted as shown in Fig. 1. The final program is a collection of the nodes at the last refinement level P_n.

The design can be probed to any desired level of detail i $(0 \leq i \leq n)$. Understanding the design process is aided by the fact that level i provides an overview of levels $i + 1$ through n.

We illustrate the stepwise refinement process with annotated examples. The notation we will use for conveying our ideas will be Pascal-like[5] and include guarded commands.[6] PL/I will be used to show the executable versions of some programs.

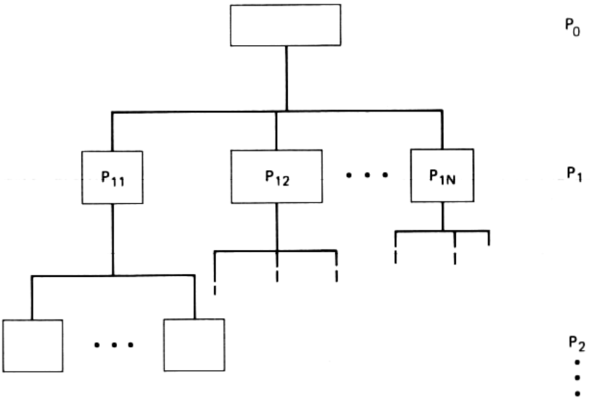


Fig. 1—The refinement process.

The guarded commands are

(*i*) *Selection*

```
if  $b_1$   →  $SL_1$
  []$b_2$   →  $SL_2$
            ⋮
  []$b_n$   →  $SL_n$
fi
```

The b_i's are called the guards (Boolean expressions) and the SL_i are statement lists. For a successful execution of the selection statement, at least one of the guards must be true. If only one guard is true, then the corresponding statement list is executed. If more than one guard is true, then one of the corresponding statement lists is selected nondeterministically (i.e., the user cannot tell beforehand) and executed, e.g.,

```
if  $a \geq b$   →   max := $a$
  []$b \geq a$   →   max := $b$
fi
```

If $a = b$, then both the guards are true and either of the statements max := a or max := b may be executed. Either way, the answer is right. This symmetry is aesthetically pleasing when compared to conventional deterministic programming.

(*ii*) *Repetition*

```
do  $b_1$   →  $SL_1$
  []$b_2$   →  $SL_2$
            ⋮
  []$b_n$   →  $SL_n$
od
```

The loop is repeatedly executed as long as one of the guards is true. If one guard is true, then the corresponding statement list is executed. As in the selection statement, if more than one guard is true, then one of the corresponding lists is arbitrarily selected and executed.

Implementation of these statements in *C*, Pascal, PL/I, etc., will be deterministic. For example, in PL/I:

(*i*) *Selection*

```
IF b1
   THEN DO; SL1; END;
   ELSE IF b2 THEN DO; SL2; END;
          ⋮
          ELSE IF bn THEN DO; SLn; END;
                    ELSE ERROR;
```

(*ii*) *Repetition*

```
L:DO WHILE ('1' B);
          IF b1
             THEN DO; SL1; END;
             ELSE IF b2 THEN DO; SL2; END;
                  ⋮
                  ELSE IF bn THEN DO; SLn; END;
                       ELSE GOTO LE;
          END L;
LE:;
```

Note: These statements could be more conveniently implemented using the new PL/I SELECT and LEAVE statements.

III. EXAMPLES OF STEPWISE REFINEMENT

The examples used to illustrate stepwise refinement are small out of necessity. The reader is encouraged to apply stepwise refinement to larger problems.

Example 1

Write a program to simulate a week in John's life.
Initial refinement P_0:
Simulate a week in John's life

If we were programming in a language that understood the above instruction, then we wouldn't have to refine it further.

Refinement P_1:

```
a.    d := monday {next day to be simulated is d}
b.    repeat
c.       simulate day d in John's life
d.       d := next day
e.       until week over
```

A refinement consists of programming language instructions mixed with English statements.

Refinement P_2:
Line c of P_1 is refined as

```
Sleep until alarm goes off
Go through morning ritual
Spend the day
Go through evening ritual
Prepare to sleep
```

Line d is refined as

```
if  $d$ = sunday    →   $d$ := monday
 []$d \neq$ sunday  →   $d$ := SUCC($d$)
fi
```

where the Pascal function SUCC gives the next day in the range of values monday, tuesday, $\cdots$, sunday. Line e: "week over" is refined as "d = monday."

Collecting these refinements of P_1's instructions, we get refinement P_2.

```
d := monday
repeat
    Sleep until alarm goes off
    Go through morning ritual
    Spend the day
    Go through the evening ritual
    Prepare to sleep
    if  $d$ = sunday    →   $d$ := monday
     []$d \neq$ sunday  →   $d$ := SUCC($d$)
    fi
    until $d$ = monday.
```

This collection can be done mechanically and we shall in general omit it.

"Spend the day" may be refined as

```
if  weekday   →   go to work
                  work
                  return home
 []weekend    →   read newspaper
                  laze around
                  read book
                  watch TV
fi
```

Similarly, the other instructions of P_2 may be refined and the refinement process continued to the desired level of detail. In the refinement we have tried to model processes of the problem domain.[7]

An initial decomposition might not be feasible or nice, in which case we back up and try another decomposition. We shall only present the final set of decompositions.

Example 2

Write a program that reads in a list of positive numbers a_1, a_2, $\cdots$, a_n ($n \geq 0$) and prints the sums of all natural numbers up to each a_i, i.e., the sums:

$$\sum_{i=0}^{a_1} i, \quad \sum_{i=0}^{a_2} i, \quad \cdots, \quad \sum_{i=0}^{a_n} i.$$

Initial refinement P_0: Print $\sum_{i=0}^{a_1} i, \sum_{i=0}^{a_2} i, \cdots, \sum_{i=0}^{a_n} i$.

P_1*:

```
read $a$
do while there
   exists data  →   Compute sum = $\sum_{i=0}^{a} i$.
                    Print sum
                    read $a$
od
```

Because we are aiming for an executable program in a sequential programming language, the refinement P_1 reflects the decision to read in an input element, compute its sum, print the sum, and then read another input element. Alternately, had our target been a parallel computer we would have probably read in all the input elements, computed the sums in parallel, and then printed them out. *Many implicit decisions underlie every refinement.*

P_2:

- while there exist data
  is refined to
  not EOF
- Compute sum = $\sum_{i=0}^{a} i$
  is refined to

```
$i$ := 0
sum := 0          {sum = 0 + 1 + 2 + ··· + $i$}
do $i \neq a$   →   $i$ := $i$ + 1; sum := sum + $i$ od
```

Let us now examine the concept of a loop invariant. A loop invariant is an assertion about program variables; it statically captures the meaning of a loop thus helping us understand it. Loop invariants are true before and after the execution of a loop, and before and after each execution of the loop body. Dijkstra[6] suggests some ways of finding the loop invariant using the desired post-condition (state of variables after the loop terminates). The loop invariant can actually aid in determining the guards and the corresponding statement lists.

Let I be the loop invariant sum $= 0 + 1 + 2 + \cdots + i$. I is true initially because $i = 0$ and sum $= 0$. Evaluation of the guard $i \neq a$ does not affect I; the statement $i := i + 1$ destroys I, resulting in sum $= 0 + 1 + 2 + \cdots + i - 1$. But sum $=$ sum $+ i$ restores the validity of invariant I. When the guard evaluates to false, i.e., $i = a$, the loop terminates. Now in addition to I being true we have $i = a$, implying the desired result sum $= 0 + 1 + 2 + \cdots + a$.

How can we demonstrate loop termination? For this we must show the existence of a function, initially ≥ 0, whose value is decreased by

* The fact that we have two read statements in P_1 shows that our design is influenced by our target language (PL/I in this case). In PL/I, unlike in Pascal, a read must occur on an empty file before an end-of-file is indicated (via the variable EOF in our case).

one every time the loop is executed. When this function becomes ≤ 0, we stop. Such a function is $a - i$; executing $i := i + 1$ decreases its value by 1. When $a - i = 0$, we have $i = a$ which is when the guard evaluates to false and the loop terminates.

Continuing the refinements we get

P_3:

```
read $a$
do not EOF  →  {compute sum = $\sum_{i=0}^{a} i$}
               $i := 0$
               sum := 0           {sum = 0 + 1 + 2 + ⋯ + $i$}
               do $i \neq a$   →   $i := i + 1$
                                  sum := sum + $i$
               od
               print sum
               read $a$
od
```

P_4 (in PL/I):

```
SUM: PROC OPTIONS(MAIN);
 DCL (A /*NEXT INPUT ELEMENT */
      ,I /*LOOP VARIABLE */
       ,SUM /*SUM=0+1+2+···+I */
       )FIXED DEC,
       EOF BIT(1) INIT('O'B);
 ON ENDFILE EOF='1'B;

  GET LIST(A);
  DO WHILE (~EOF);
     I=0; SUM=0;
      DO WHILE (I~=A);
         I=I+1; SUM=SUM+I; END;
     PUT SKIP LIST (' SUM UPTO', A, ' IS ', SUM);
     GET LIST(A);
     END;
 END SUM;
```

Example 3

Write a program to determine the maximum element value of an $m \times n$ array A ($m, n \geq 1$).

```
$P_0$: determine the max element value of $A$
$P_1$: $i$:=0     {last row examined}
       initialize max   {max is the maximum element
                         of rows 1 ⋯ $i$—loop invariant $I_1$}
```

```
    do all rows          →   $i := i+1$
       not examined          max:=maximum(row $i$, max)
    od
P2: • initialize max is refined to
      max := $A[1, 1]$
    • all rows not examined
      is refined to
      $i \neq m$
    • max := maximum (row $i$, max)
      is refined as
      $j = 0$   {max = maximum of rows 1 . . . $i - 1$ and elements
                   1 . . . $j$ of row $i$—loop invariant $I_2$}
      do all elements of        →   $j := j + 1$
          row $i$ not examined          max := MAX(max, $A[i, j]$)
      od
```

I_2 is the loop invariant. As an exercise, the reader should try and show that the loops leave I_1 and I_2 invariant, i.e., unchanged.

```
P3: • all elements of row $j$ not examined
      is refined as
      $j \neq n$
    • max := MAX(max, $A[i, j]$)
      is refined as

            if  max $\geq A[i, j]$   →   skip
             []max $\leq A[i, j]$   →   max := $A[i, j]$
            fi
```

where skip denotes the null statement.

The iterative feature is the most important feature of a programming language.[8] The **do** ··· **od** construct allows us to express algorithms clearly and succinctly. The above example could have been done better had the author not used the **do** ··· **od** construct to just simulate the **while** statement. Making fuller use of the **do** ··· **od** construct, we get the following program for the above problem:

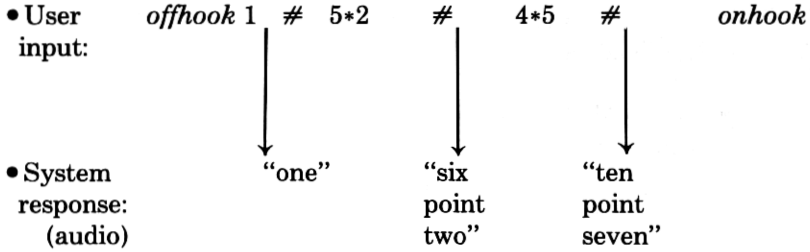
```
P'2: $i := 0$   {number of rows examined so far}
     $j := 0$   {number of elements of row $i + 1$ examined so far}
     initialize max {max is the maximum of all the elements in the first
                     $i$ rows and the first $j$ elements of row $i + 1$} – $I$
     do  $i < m$ rows and $j < n$     →   $j := j + 1$
         elements of row $i + 1$          max := MAX(max, $A[i, j]$)
         examined
       []$i < m$ rows and all         →   move to the next row
         elements of row $i + 1$
         examined
     od
```

```
$P'_3$:  $i := 0$;   $j := 0$
         max := $A[1, 1]$   {max is the maximum of all the
                              elements in the first $i$ rows
                              and the first $j$ elements of row $i + 1$} − $I$
         do  $i < m$ and $j < n$   →   $j := j + 1$
                                       max := MAX(max, $A[i, j]$)
           []$i < m$ and $j = n$   →   $i := i + 1$; $j := 0$
         od
```

Gries also shows that the **do** ··· **od** construct usually eliminates the need for loop exits necessary in programs that use the **while** statement.[8]

Example 4

The Touch-Tone® telephone provides an easy but limited means of communicating with a computer (see Fig. 2). The problem is to write a program that provides a simple adding machine to the user.[9] For example:


• User input:
offhook 1 # 5∗2 # 4∗5 # *onhook*
• System response: (audio)
"one"
"six point two"
"ten point seven"

The characters # and ∗ represent + and ·, respectively.

The following modules are available to the programmer:

- SPEAK (string)—provides an audio response for the number represented by the string.

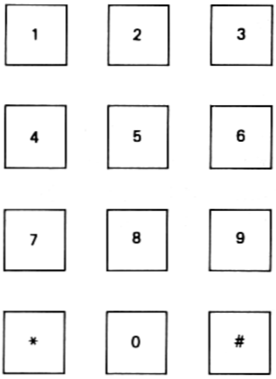


Fig. 2—The Touch-Tone® telephone's pushbutton dial.

1	·	5	3	null

- ADD(string1, string2) – string1 := string1 + string2

		8	·	3	null	string 1
+		4	null			string 2
	1	2	·	3	null	string 1

- waitsignal(char)—sets char to the next input character when available.

The input/output specifications written more formally are:

input: *offhook* f_1 # f_2 # $\cdots$ # f_n # *onhook*,

where $\underset{(1 \le i \le n)}{f_i} = \begin{cases} \text{integer} & \text{one or more digits} \\ \text{real} & \text{—one or more digits followed by} \\ & \text{—one or more digits followed by } * \text{ and at least one digit} \\ & \text{—one } * \text{ followed by at least one digit} \end{cases}$

output: SPEAK(SUM_1), SPEAK(SUM_2), $\cdots$, SPEAK(SUM_n)
where $\text{SUM}_i = \sum_{k=1}^{i} f_k$, $1 \le i \le n$,
and the audio response occurs after the character # is input.

We assume that the maximum length of numbers input will be $k - 1$. To focus on the refinement process, we make the following additional assumptions:

(*i*) one addition session,
(*ii*) no errors of any kind.

In the second version of the solution we will eliminate these restrictions.

Refinement P_0: Do telephone addition.

```
P1:    Compute and speak out the running sum of the numbers
                           input.
P2:    plus := '#';   point := '*'
       waitsignal(c) {c contains the next
                      input char to be processed;
                      offhook is the first one}
       waitsignal(c) {get char after offhook}
       initially SUM is 0
       do c ≠ 'onhook'   →   read number into A
                             ADD(SUM, A)
                             SPEAK(SUM)
                             waitsignal(c) {+ consumed}
       od
```

The number variables SUM and *A* are implemented as strings because modules SPEAK and ADD expect strings as arguments.

Each variable is represented by

(*i*) variable of type string,

(*ii*) an integer variable that denotes the length of the string,

where **type** string = **array** [1 ··· *k*] **of** char.

In addition to SPEAK and ADD we need

(*i*) a procedure ZERO to initialize the numbers represented as strings to 0,

(*ii*) a procedure APPEND to help build a number.

They are defined as

```
procedure ZERO(var X:string; l:integer);
   begin l:=1; X[l]:=null end
procedure APPEND(var X:string; l:integer; c:char);
   begin X[l]:=c; l:=l + 1; X[l]:=null end
```

The implementation of numbers and their operations in terms of strings is an example of *data refinement*.

- Read number into *A* is refined as

```
ZERO A
do c ≠ plus  →  if c = point  →  APPEND '·' to A
                   [] c ≠ point  →  APPEND c to A
                fi
                waitsignal(c)
od
```

Collecting the refinements together we get

```
const point = '*'; plus = '#';
type string = array[1 ·· k] of char;
var SUM, A:string;
   lSUM, lA:integer; {lengths of SUM, A}
begin waitsignal (c); waitsignal (c);
   ZERO(SUM, lSUM);
   do c ≠ 'onhook'  →  {read number into A}
          ZERO(A, lA)
          do c ≠ plus
             if c = point  →  APPEND(A, lA, '·')
             [] c ≠ point  →  APPEND(A, lA, c)
             fi
             waitsignal(c)
          od
          ADD(SUM, A);
          SPEAK(SUM);
          waitsignal(c)
```

```
od
end
```

Instead of including inline the refinement of “read number into A” in the final version of the program, it would have been more appropriate to make the refinement into a procedure READ and to call READ from the final version. This is because READ and the operations ADD and SPEAK (which appear in the final program) operate on numbers thus representing the same level of abstraction. Also, if an instruction appears more than once, then it should perhaps become a procedure call. The instruction would then be refined only once.

In the following version of the above program we eliminate the restrictions of a single session and no errors. The following types of errors are considered possible:

- illegal characters,
- $n \geq 2$ decimal points per number,
- only a decimal point—no digits,
- + follows +, *offhook*, i.e., null number,
- session starts with other than *offhook*.

The initial problem formulation P_0 is

```
            do true  →  Do telephone addition od
```

P_1: • Do telephone addition is refined as
Compute and speak out the running sum of the numbers input so far.

This is refined as

P_2: waitsignal(c)

```
  if $c \neq$ 'offhook'  →  error
  [] $c =$ 'offhook'     →  skip
  fi
  waitsignal(c)
  check for valid char {digits, plus, point, 'onhook'}
  initially sum is 0
   do $c \neq$ 'onhook'  →  Read number into A
                            if $c =$ plus  →  ADD(SUM, A)
                                              SPEAK(SUM)
                                              waitsignal(c)
                                              check for valid char
                            [] $c =$ 'onhook' → skip
                            fi
   od
```

Using the procedures APPEND and ZERO defined before, we refine read number into *A* as

```
ZERO A
# digits, # pts := 0;
do c is a
    digit or a point
                      → if c is a point → Append '·' to A
                                           if # pts = 0 → # pts := # pts + 1
                                           [] # pts ≠ 0 → error
                                           fi
                        [] c is a digit → Append c to A
                                           # digits := # digits + 1
                        fi

    waitsignal(c);
    check for valid char
od
if # digits = 0 → error
[] # digits ≠ 0 → skip
fi
```

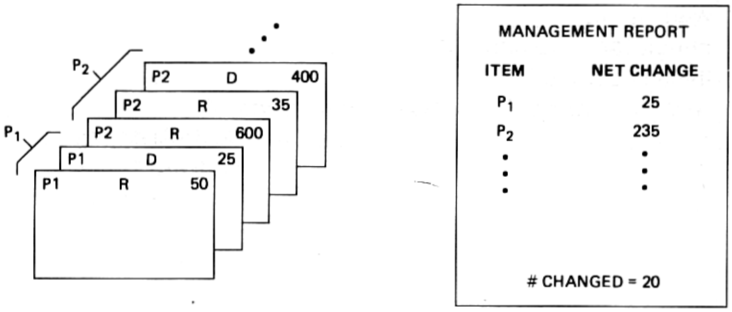
Continuation of this refinement process is similar to the error-free version and we omit it.

Example 5. McDonald's warehouse problem[9]

Given a list of item cards ordered by item number, produce the management report shown in Fig. 3. Each invoice has an item number, a code *D* for delivery, and *R* for received, and the quantity received or delivered.

P_0: Produce management report

P_1: a. Print heading
 b. Process the item groups
 c. Print number of item groups changed



MANAGEMENT REPORT

ITEM	NET CHANGE
P_1	25
P_2	235
⋮	⋮

CHANGED = 20

Fig. 3—From item cards to management report.

- Line b is refined as

```
# changed := 0
read item
do there are more → process an item group
    item groups      print item and net change
                     # changed := # changed + 1
od
```

- there are more item groups
  is refined as
  not EOF
- process an item group
  is refined as

```
netchange := 0
itemgroup # := item#
do item in group and not EOF
    → if code = R → netchange := netchange + Qty
       [] code = D → netchange := netchange − Qty
       fi
       read item
od
```

- item in group
  is refined as

```
itemgroup # = item#
```

This concludes the example.

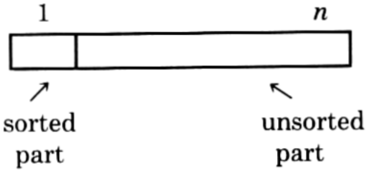
Example 6

Using insertion sort, sort the array A (size $n \geq 1$) in nondecreasing order, i.e., $A_1 \leq A_2 \leq \cdots \leq A_n$, and the new values of array A are a permutation of its old values.

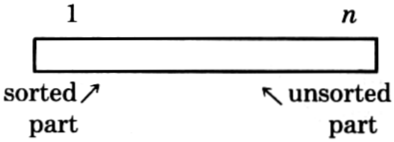
P_0: Sort the array A

Pictorially we can characterize the input and output specifications of the array A as

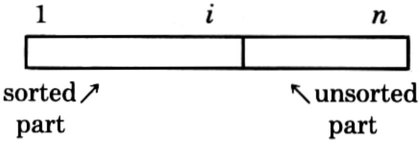
(i) initially



(*ii*) finally



At some intermediate stage of the sorting we will have



This picture corresponds to our loop invariant. It is true initially if $i = 1$. At the end of the loop, $i = n$ implies that the whole array is sorted. So the purpose of the loop body will be to exchange the values of A in such a way that i can be increased until it equals n. P_0 is therefore refined as

P_1:

```
$i := 1$     {$A[1 \cdot\cdot i]$ sorted}
do $i \neq n \rightarrow$ Extend sorted portion to include $A[i + 1]$
               $i := i + 1$
od
```

- Extend sorted portion to include $A[i + 1]$ is refined as

```
a. $t := A[i + 1]$
b. shift all elements of $A[1 \cdots i] > t$
   one place to the right such that
   $A[1 \cdots j - 1] \leq t$ and $A[j + 1 \cdots i + 1] > t$
c. $A[j] := t$
```

- Line *b* of the above refinement is developed as

```
$j := i + 1$ {$A[j + 1 \cdots i + 1] > t$}
do $A[j - 1] > t \rightarrow$ shift $A[j - 1]$ to the right
                  $j := j - 1$
od
```

On loop termination we have $A[j + 1 \cdot\cdot i + 1] > t$ and $A[j - 1] \leq t$. This, along with the fact that at the start $A[1 \cdots i]$ was sorted, leads us to $A[1 \cdots j - 1] \leq t$.

As we have not taken proper care of the end condition, the guard in the above loop will cause a subscript error when $j = 1$. So we modify it to

$$j \neq 1 \textbf{ cand } A[j - 1] > t$$

where **cand** is similar to **and** but the second operand is evaluated only if $j \neq 1$ (C has a similar operator). This allows for a simple high level design.

- shift $A[j-1]$ to the right is refined as

$$A[j] := A[j-1]$$

Collecting all the refinements we get

```
i := 1
do i ≠ n → t := A[i + 1]; j := i + 1
             do j ≠ 1 cand A[j − 1] > t → A[j] := A[j − 1]
                                          j := j − 1
             od
             A[j] := t
             i := i + 1
od
```

We conclude this section with some comments on program maintenance and efficiency. Program maintenance, i.e., program modification that is due to changing specifications or in response to design errors, should be carried out by making changes in the refinements and not just the final program. Making changes in only the final program renders the design (i.e., refinements) obsolete; consequently, an updated version of the design will no longer exist and subsequent program maintenance becomes increasingly difficult.

When program specifications change, start from the initial refinement and locate the refinement affected. Modify this refinement and carry the effects of this change down to the last level of refinement.

When a design error is detected, locate the most abstract refinement in which the design error was first made. Then carry the change caused by the removal of the design error down to the final refinement.

A program that does not have the desired efficiency (i.e., performance) characteristics must be redesigned. We locate the most abstract refinement R where a design decision was made that resulted in these characteristics. A proper design modification from refinement R onwards leads to the desired efficiency characteristics. Predecessors of refinement R remain unchanged.

IV. RECURSION

Stepwise refinement and recursion blend naturally with each other. Many programmers avoid recursion and treat it as a novelty.[10] Some problems are best expressed recursively, even though languages like FORTRAN and COBOL are not recursive, and this inhibits programmers from thinking and designing recursively. Also, the examples of recursion in text books, e.g., factorial, Fibonacci numbers, etc., are not convincing about its utility.

Efficiency has often been cited as a reason against using recursion. In these days of increasing software costs and decreasing hardware costs, this reason is not very convincing. It is better to have a recursive design that is simpler, easier to understand, and easier to show correct than a corresponding nonrecursive version. If efficiency is still a criterion, then the recursive design can be systematically transformed into a nonrecursive one.[11] The following examples illustrate the development of recursive programs:

1. Write a procedure to print a binary tree with root R (Fig. 4). Each node is of the form

<table>
<tr><td colspan="2">VALUE</td></tr>
<tr><td>LEFT</td><td>RIGHT</td></tr>
</table>

where

(*i*) VALUE is the data at the node,
(*ii*) LEFT, RIGHT are the pointers to the subtrees,
(*iii*) a NIL pointer value denotes the absence of a subtree.
Initial refinement P_0: Print binary tree with root R

P_1:

```
        if $R \neq$ NIL   →   Print binary tree with root LEFT($R$)
                                          {left subtree}
                              Print VALUE($R$)
                              Print binary tree with root RIGHT($R$)
                                          {right subtree}
        []$R =$ NIL   →   skip
        fi
```

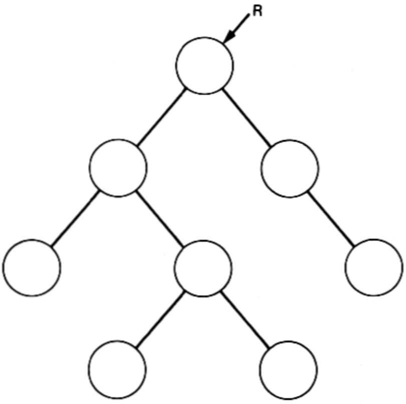


Fig. 4—A binary tree with root R.

The notation $p(x)$ denotes the component p of the node pointed to by x. Writing the above in Pascal we get:

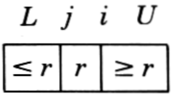
```
procedure print (R: ↑ node);
  begin
    if R ≠ nil
      then begin print(R ↑. LEFT);
                 writeln(R ↑. VALUE);
                 print(R ↑. RIGHT)
                 end
      end
```

2. This example illustrates a fast sorting technique called quicksort (Hoare[12]). Array A, with bounds L and U ($L \leq U$), is to be sorted in nondecreasing order.

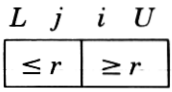
Initial refinement P_0: Quicksort(A, L, U).

P_1:

```
if one element              →  skip
[ ]two elements             →  order them
[  ]more than two elements  →  Partition A such that
```



or



```
                                (at least one element per partition
                                in this case)
                                where r is an arbitrary value
                                Quicksort(A, L, j)
                                Quicksort(A, i, U)
fi
```

```
P2: • one element
      is refined as
      U − L = 0
    • two elements is refined as U − L = 1
    • order them
      is refined as
            if A[U] ≤ A[L]  →  swap(A[L], A[U])
            [] A[U] ≥ A[L]  →  skip
            fi
    • more than two elements
      is refined as
      U − L > 1
```

- Partition A such that ···
  is refined as

```
$r := A[(U + L) \div 2]$
$i := L; j := U$ $\{A[L .. i - 1] \le r$ and $A[j + 1 .. U] \ge r$ – Invariant $I\}$
do $i < j$  →  Extend left partition by increasing $i$
               Extend right partition by decreasing $j$
               Rearrange elements so that invariant $I$ is
                 restored
od
```

```
$P_3$:   • Extend left partition ···
         do $A[i] < r$  →  $i := i + 1$ od $\{A[i] \ge r\}$
         • Extend right partition ···
         do $A[j] > r$  →  $j := j - 1$ od $\{A[j] \le r\}$
         • Rearrange ···
             if $i < j$  →  swap $(A[i], A[j])$
                            $i := i + 1; j := j - 1$
               [] $i \ge j$  →  skip
             fi
```

V. MULTIVERSION PROGRAMS

A set of programs is said to constitute a program family if it is worth while to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members. A typical family is the set of versions of an operating system distributed by a manufacturer.[13] Such a family is also called a set of multiversion programs. Stepwise refinement enables multiversion programs to be developed conveniently and naturally.

Multiversion programs may be built for the following reasons:[13,14]

(*i*) Economics. It is cheaper to build one program and then modify it to get another version than it is to build the second program from scratch.

(*ii*) Experimentation. Experimental prototypes may be built to study the feasibility of building a particular system. The experimental versions along with the final program constitute the multiversions.

(*iii*) Faulty program design. Another version of the program is built to correct the design faults of a prior version.

Classically, multiversion programs have been built by first building one working version of a program. Another version is built by modifying this program and so on, as shown in Fig. 5. A set of multiversion programs produced as in Fig. 5 has one common ancestor. According to Parnas,[13] it is common for the descendants of one program to share some of their ancestors' characteristics which are not appropriate to the descendants. In building the earlier version, some decisions were

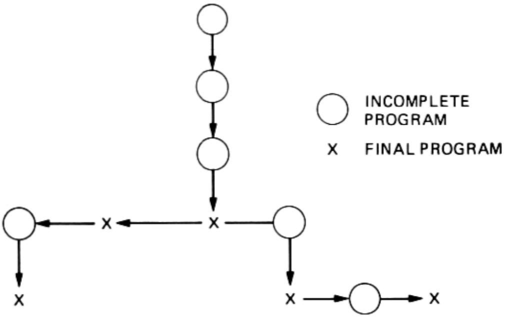


Fig. 5—Traditional way of building multiversion programs.

made which would not have been made in the descendant programs had they been built independently. Removal of these decisions entails a lot of reprogramming. Consequently, programs have performance deficiencies because they contain decisions not really suitable for them. To build another program version, the program must first be complete and working. Relevant changes in an ancestor program that are not reflected in the descendant program cause maintenance problems.

Stepwise refinement allows us to develop multiversion programs without the above problems. Never modify a complete program; always begin from one of the intermediate refinements which does not contain any design decisions unsuitable for the new version. This process is illustrated in Fig. 6.

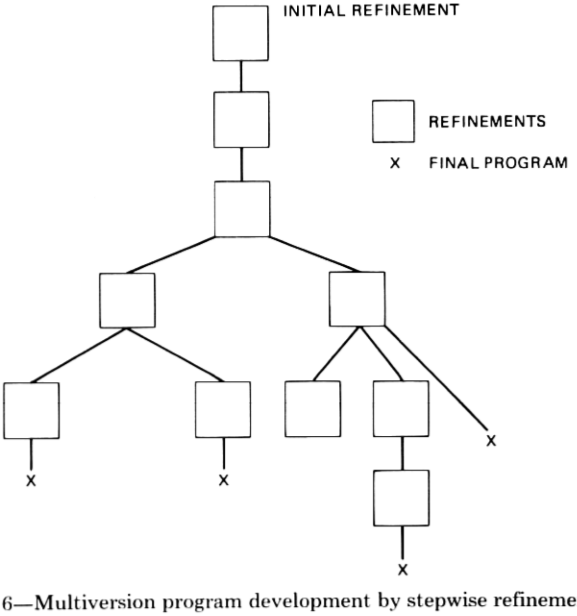


Fig. 6—Multiversion program development by stepwise refinement.

All decisions made above a branch point are shared by the descendants. Refinements are developed so that common decisions of multiversion programs are above the branch point. A branch point results when two or more versions require different strategies (e.g., storage management techniques).

Also refinements below a branch point may be carried out in parallel—we do not have to wait until a working program is available.

VI. SUGGESTIONS FOR REFINEMENT

1. Develop the program in a gradual sequence of steps.

2. In each step, refine one or more instructions of the given refinement.

3. Terminate the refinement process when the instructions have been expressed in the desired programming language or when they can be mechanically translated to the programming language.

4. Use information about the problem and its domain in the formulation of abstract instructions.

5. Use notation natural to the problem domain.

6. Make up abstract instructions as desired. However, they must eventually be translatable to an executable form.

7. Make refinements reflect the instructions they represent in detailed form.

8. Be aware that every refinement represents some implicit design decision and consider alternate solutions. Keep a written record of the major decisions made along with the refinements.

9. Use recursion when appropriate. Even if the language does not support recursion, recursive solutions should still be considered. If recursive solutions are selected, they can be systematically converted to nonrecursive solutions.

10. Use data refinement along with instruction refinement.

11. Postpone representation of data as long as possible. This minimizes modifications to the design when an alternate representation is to be used.

12. If an instruction appears more than once, use a procedure call; refine the instruction only once. Use procedure calls when they clarify program structure.

13. Recognize abstract data types and separate their refinements from the rest of the program, i.e., do not refine the data type operations in line—use procedure calls.

14. Try to use loop invariants to develop loops; they give a better idea of the instructions in the loop body and the guards.

15. If a refinement solution does not turn out to be appropriate, repeat the refinement process using the additional knowledge derived from the previous attempt; stepwise refinement is an iterative process.

VII. TOPICS RELATED TO STEPWISE REFINEMENT

In this section, the idea of abstract data types is explained along with the concept of data refinement. This is followed by a discussion of the formal specifications of abstract data types. We then show how stepwise refinement may help simplify program correctness proofs. Finally, we briefly argue that stepwise refinement can be used in developing parallel programs.

7.1 Abstract data types

A data type is not only a set of values but also the operations that can be performed on them.[15] A primitive data type is a data type that is available in the programming language. An abstract data type is a data type not available in the programming language and is implemented in terms of other abstract and primitive data types.

The implementation of a data type consists of three parts—storage allocation, initialization, and the definition of operations. Consider the following implementation of the data type integer stack of size 100 in PL/I:

(*i*) storage allocation

```
        DCL (S(100)
            ,NS /*NUMBER OF ELEMENTS IN S*/
            )FIXED BIN;
```

(*ii*) initialization

```
                        NS = 0;
```

(*iii*) operations

```
    PUSH:PROCEDURE(A, NA, X);
         DCL (A(100), NA, X) FIXED BIN;
         IF NA = 100
            THEN PUT SKIP LIST ('OVERFLOW ERROR');
            ELSE DO; NA = NA + 1; A(NA) = X; END;
         END PUSH;

    POP:PROCEDURE(A, NA);
        DCL (A(100), NA)FIXED BIN;
        IF NA = 0
           THEN PUT SKIP LIST ('UNDERFLOW ERROR');
           ELSE NA = NA - 1;
        END POP;

  TOP:PROCEDURE(A, NA) RETURNS (FIXED BIN);
      DCL (A(100), NA)FIXED BIN;
      IF NA = 0
         THEN PUT SKIP LIST('ERROR-STACK EMPTY');
         ELSE RETURN(A(NA));
      END TOP;
```

```
EMPTY: PROCEDURE(A, NA)RETURNS(BIT(1));
       DCL (A(100), NA)FIXED BIN;
       RETURN(NA = 0);
       END EMPTY;
```

Such implementations of abstract data types suffer from many disadvantages. Representation details are not hidden from the programmer. This leads to the following:

(*i*) Representation dependent programming, perhaps for "efficiency" reasons. For example, the programmer may directly inspect the top of the stack instead of using the procedure TOP; a change in the representation will then require changes in the program.

(*ii*) Violation of the specifications of the abstract data type. For example, in case of the stack the programmer may delete an element in the middle of the stack. If such an ability is desired, then the specifications should be changed.

(*iii*) Inadvertent or malicious violation of the integrity of the abstract data type. For example, *NS* could be set to 0 even if the stack is not empty.

In addition, the programmer has to be concerned about which components of the representation have to be passed as arguments to the procedures associated wth the abstract data type. To be uniform, we have passed all the components of the stack representation for every operation.

Modern programming languages such as CLU[16–18] and Alphard[1] provide data abstraction features, called clusters and forms, respectively, *that remove the above problems*. In addition, they support *data refinement*. We use CLU's clusters to illustrate data refinement and give an example of programming with abstract data types. The notation used is similar to that of CLU.

```
stack = cluster is push, pop, top, empty;
  rep = record [ns:integer;
        s: array[1 ·· 100] of integer]
  create = oper(   ) returns cvt;
      s:rep
      s.ns:=0
      return s
      end
  push = oper(a:cvt, x:integer)
    if a.ns = 100
    then overflow error
    else begin a.ns := a.ns + 1
               a.s[a.ns] := x
               end
```

```
          end
      pop = oper(a:cvt)
        if a.ns = 0
              then underflow error
                  else a.ns := a.ns − 1
          end
      top = oper(a:cvt) returns integer
          if a.ns = 0
                then stack empty error
                else return a.s.[a.ns]
          end
      empty = oper(a:cvt) returns boolean
            return a.ns = 0
            end
      end stack
```

Having defined the stack cluster, the programmer can declare variables of type stack, e.g., *b:stack*.

The first line of the stack definition states that the operations available to stack users are push, pop, top, and empty. The operation must be prefixed by the type name, e.g., the special operation **create** is automatically executed when a variable of type stack is declared.

The line beginning "**rep** = " specifies that a stack is represented by an integer array and an integer. This information cannot be used outside the cluster, thus making the rest of the program representation independent.

The special symbol **cvt** (convert) means that the variable is of the abstract type outside the operation and of the representing type inside the operation.

An operating system example

Suppose we are writing an operating system. Jobs are to be scheduled according to their priority (10 being the highest and 1 the lowest). The next job to be executed is the one with the highest priority. If there is more than one job with the highest priority, then the one selected for execution is the one that waited the most (FIFO):

```
begin {operating system}
      .
      .
      .
      Add job j with priority p to the list of
            jobs waiting for execution
      .
      .
      .
      Wait until there is a job to execute
      let j be the next job to be executed
      .
      .
      .
      end {operating system}
```

To implement the job scheduling we define a cluster called spq (system of priority queues), with operations add, empty , and next job.

```
spq = cluster is add, empty, nextjob
  rep = array[1 ·· 10] of queue
  create = oper(   ) returns cvt
      s:rep
      return s
      end
add = oper(s:cvt, job:integer, prty:1 ·· 10)
    queue$add(s[prty], job≠)
    end
empty = oper(s:cvt) returns boolean
    i: integer := 0
    while i ≠ 10 do
         begin i := i + 1
         if queue$empty(s[i]) then return false
         end
    return true
    end

nextjob = oper(s:cvt) returns integer
    i:integer := 11
    j:integer
    while i ≠ 1 do
        begin i := i − 1
              if ~ queue$empty(s[i])
                then begin j := queue$front(s[i])
                        queue$delete(s[i])
                        return j
                        end
            end
    end
end spq
```

The cluster spq is implemented in terms of the abstract data type queue. For example, cluster spq's operation add uses cluster queue's operation add, i.e., queue$add. We refine instructions of the abstract data type queue by implementing a queue cluster. For this example, we assume that no more than 50 jobs of the same priority will be waiting at the same time. We shall use a wrap-around array representation for the queue in which

(*i*) an array of size 51 is used; only 50 elements can be stored in the queue,

(*ii*) $F = L$ means that the queue is empty,

(*iii*) **mod**$(F, 51) + L$ points to the next element in the queue,
(*iv*) **mod**$(L, 51)$ points to the last element in the queue,
(*v*) **mod**$(L, 51) + 1 = F$ means that the queue is filled.

```
  queue = cluster is delete, add, front, empty;
          rep = record[a:array[1 ·· 51] of integer; F, L:integer]
          create = oper(   )returns cvt
                 q:rep
                 q.F := 0
                 q.L := 0
                 end

  delete = oper(q:cvt);
          if q.F = q.L
                      then error-empty queue
                      else q.F := mod(q.F, 51) + L
                  end
  add = oper(q:cvt, j:integer)
          if mod(q.L, 51) + 1 = q.F
                  then error-queue full
                  else begin q.L = mod(q.L, 51) + 1
                             q.a[q.L] := j
                             end
          end

front = oper(q:cvt) returns integer
      if q.F = q.L
          then error-empty queue
          else return q·a[mod(F, 51) + 1]
      end

empty = oper(q:cvt) returns boolean
      return q.F = q.L
      end
end queue
```

7.2 Formal specifications of abstract data types

In this section, we consider the formal specifications of abstract data types. In particular, we take a brief look at the algebraic specification technique proposed by Guttag.[19, 20] Abstract data types are implemented in terms of other data types by the refinement or decomposition of specifications.

The algebraic specifications consists of two parts:

(*i*) the syntax—here the operations of the data type are listed indicating the number of arguments, the argument types, and the result type.

(*ii*) the semantics—here axioms are given that relate the values created by the operations.

In the basic notation which we will use, the operations are functions without side effects; none of the arguments are changed. Guttag[21] has extended the notation to allow for changes in arguments, i.e., to allow procedures.

We shall present algebraic specifications for the abstract data type stack considered earlier. We shall then give specifications for an array. Finally we shall refine the stack operations in terms of array operations. To keep the specifications as simple as possible, we shall consider unbounded (i.e., infinite size) stacks and arrays.

Stack specifications:

```
1.        type stack
2.            syntax
3.                  create( )  →  stack
4.                  push(stack, integer)  →  stack
5.                  pop(stack)  →  stack
6.                  top(stack)  →  integer
7.                  empty(stack)  →  boolean
8.            semantics
9.                  declare s:stack; x:integer
10.                 pop(create( )) = underflow error
11.                 pop(push(s, x)) = s
12.                 top(create( )) = empty stack error
13.                 top(push(s, x)) = x
14.                 empty(create( )) = true
15.                 empty(push(s, x)) = false
```

Line 3: specifies the syntax of the create operation. The result of calling create, which has no parameters, is an object of type stack.

Line 4: operation push takes as input a parameter of type stack and a parameter of type integer. It returns a value of type stack.

Line 10: The result of applying the pop operation on a stack that has just been created is an underflow error. "=" is the equality operator (not assignment).

Line 11: The result of popping a stack *s* on which the last operation was to push a value *x* is the initial stack *s*.

These specifications specify the abstract data type completely. For details on how to construct the specifications, see Ref. 20. We now give the specification for the type array which will be used to implement the type stack.

```
1.      type array
2.          syntax
```

```
            createarray( )  →  array
            assign(array, integer, integer)  →  array
            access(array, integer)  →  integer
        semantics
            declare a:array; i, j, x:integer
            access(createarray( ), i) = undefined error
            access(assign(a, i, x), j)
                    = if i = j
                        then x
                        else access(a, j)
```

Operation assign(a, i, x) stands for an array whose ith element has been assigned the value x. Operation access(a, i) stands for the ith element of a.

We now implement the abstract data type stack by decomposing stack operations in terms of array operations. We give axioms called programs that give the effect of stack operations in terms of array operations.

```
    stack implementation
        syntax
            STK(array, integer)  →  stack
        programs
            declare a:array, t, x:integer
            create( ) = STK(createarray( ), 0)
            push(STK(a, t), x) = STK(assign(a, t + 1, x),
                                                t + 1)
            pop(STK(a, t)) = if t = 0
                                then underflow error
                                else STK(a, t − 1)
            top(STK(a, t)) = if t = 0
                                then empty stack error
                                else access(a, t)
            empty(STK(a, t)) = (t = 0)
```

Given the formal specifications for an abstract data type, an initial inefficient implementation, called the direct implementation, can be automatically generated.[19] Thus one can test some facets of a high-level data type before fixing upon a particular implementation. Thus a true top-down implementation methodology can be achieved.

7.3 Program correctness

Stepwise refinement provides a natural environment for reducing the problem of showing the correctness of a large program into showing the correctness of several smaller programs.

A program is said to be correct if it meets its input and output specifications (which may include performance criteria). Alternately, a correct program is one that transforms a state (i.e., data values) representing the input specifications into one representing the output specifications. A program is thus viewed as a specification transformer (Dijkstra[6] calls it a predicate transformer). Let

(*i*) I and O be the input and output specifications for the problem being solved.

(*ii*) P_0 be the initial problem formulation and P_0^* the corresponding final program.

Then we say that P_0 has been solved correctly if P_0^* is correct with respect to I and O. If P_0 is a nontrivial problem, then proving the correctness of P_0^* will be correspondingly nontrivial.

Stepwise refinement allows the correctness proof of a program to be reduced to the correctness proofs of smaller programs. Suppose P_0 is refined or decomposed into the subproblems $P_{10}, P_{11}, \cdots, P_{1n}$, with each P_{1i} having specifications S_i and S_{i+1} ($S_0 = I$ and $S_{n+1} = 0$). The problem of proving P_0^* correct is now reduced to proving P_{1i}^* ($0 \le i \le n$) correct, where P_{1j}^* is the program corresponding to P_{1j}. Owicki[22] provides an example of such a correctness proof.

In summary, stepwise refinement provides a natural medium for a difficult proof to be decomposed into several smaller proofs. A proof is any convincing demonstration of a program's correctness. However, the conventional approach to understanding programs in terms of how computers execute them is inadequate. A more mathematical approach is needed even if it is used informally.[23] Alagic[24] contains many examples of programs designed with correctness proofs in mind.

7.4 Parallel programs

The development of parallel programs is no different than the development of sequential programs as far as stepwise refinement is concerned. Instead of using only sequential constructs, like **begin** S_1, S_2; $\cdots$; s_n **end** in Pascal, we now use constructs for parallel programming,[25, 26] as shown below:

(*i*) **cobegin** $S_1, S_2, \ldots, S_n$ **coend**

The statements $S_1, S_2, \ldots, S_n$ are executed in parallel.

(*ii*)

```
when $b_1$ → $SL_1$
    []$b_2$ → $SL_2$
        ⋮
    []$b_n$ → $SL_n$
    end
```

Wait till one of the guards b_i is true and then execute the corresponding statement list

```
(iii) cycle $b_1 \rightarrow SL_1$
          $[]b_2 \rightarrow SL_2$
             ⋮
          $[]b_n \rightarrow SL_n$
          end
```

Endless repetition of a when statement.

If several guards are true within a **when** or a **cycle** statement, then one of the corresponding statement lists is executed nondeterministically.

VIII. CONCLUSIONS

In this tutorial, we have tried to illustrate the stepwise refinement technique, its advantages, and related topics. Stepwise refinement can be learned easily with some practice. It blends in naturally with the newer concepts in programming languages and methodology (e.g., abstract data types, parallel programming, etc.).

Stepwise refinement does not provide a solution to the problem. No methodology, old or new, is going to discover algorithms (i.e., problem solutions) for the programmer. The algorithms must come from the programmer's education, experience, and ingenuity.

Stepwise refinement encourages the development of a problem solution in a systematic fashion that is easy to understand, modify, and improve upon. The various refinements should not be discarded once the final program version is arrived at. They are part of the program documentation. Understanding the final program without them is hard even if the program is small (e.g., the eight-line final program version of insertion sort in Section III).

The reader is urged to try stepwise refinement on some problems, especially large ones.

IX. ACKNOWLEDGMENTS

I am very grateful to the following people for their constructive and critical comments, which were very valuable. They are (in alphabetical order) A. P. Boysen, Jr., R. H. Canaday, D. G. Dzamba, A. R. Feuer, T. B. Muenzer, and D. A. Nowitz.

REFERENCES

1. W. A. Wulf, "Alphard: Toward a Language to Support Structured Programs," Computer Science Dept. report, Carnegie-Mellon University, Pittsburgh, Penn., April 1974.
2. F. P. Brooks, *The Mythical Man-Month*, Reading, Mass.: Addison-Wesley, 1975.
3. C. Alexander, *Notes on the Synthesis of Form*, Boston; Harvard University Press, 1970.

4. N. Wirth, "Program Development by Step Refinement," Commun. ACM, *14*, No. 4 (1971).
5. K. Jensen and N. Wirth, *Pascal User Manual and Report*, New York: Springer, 1974.
6. E. W. Dijkstra, *A Discipline of Programming*, New Jersey: Prentice Hall, 1977.
7. M. A. Jackson, "Information Systems: Modelling, Sequencing, and Transformations," Proc. Third Int. Conf. Software Engineering, Atlanta, Ga., 1978.
8. D. Gries, "A Note on Iteration," TR77-323, Department of Computer Science, Cornell University, Ithaca, NY.
9. G. D. Bergland, private communication.
10. D. Gries, "Recursion as a Programming Tool," TR75-234, Department of Computer Science, Cornell University, Ithaca, NY.
11. S. Sickel, "Removing Redundant Recursion," Technical Report, Information Sciences, University of California, Santa Cruz, Calif., 1978.
12. C. A. R. Hoare, "Quicksort," Comput. J., *5*, No. 1 (1962).
13. D. Parnas, "On the Design and Development of Program Families," IEEE Trans. Software Eng., March 1976.
14. R. H. Canaday, private communication.
15. J. H. Morris, Jr., "Types are Not Sets," ACM Symp. Princ. Prog. Lang., Boston, Mass., 1973.
16. B. Liskov and S. Zilles, "Programming with Abstract Data Types," Proc. SIGPLAN Symp. Very High Level Lang., Santa Monica, Calif., 1974.
17. B. Liskov, "A Note on CLU," Computation Structures Group Memo 112, MIT Project MAC, Cambridge, Mass., November 1974.
18. B. Liskov et al., *CLU Reference Manual*, Cambridge, Mass.: MIT Laboratory for Computer Science, 1978.
19. J. V. Guttag et al., "Abstract Data Types and Software Validation," Commun. ACM, *21*, No. 12 (1978).
20. J. V. Guttag, "The Algebraic Specification of Abstract Data Types," Acta Inform., *10* (1978).
21. J. V. Guttag et al., "Some Extensions to Algebraic Specifications," Proc. Lang. Design for Reliable Software, March 1977.
22. S. Owicki, "The Specification and Verification of a Network Mail System," CSL TR-159, Computer Science Lab. Report, Stanford, California (1979).
23. D. Gries, "On Believing Programs to be Correct," Commun. ACM, *20*, No. 1 (1977), pp. 49, 50.
24. S. Alagic and M. A. Arbib, *The Design of Well-Structured and Correct Programs*, New York: Springer, 1978.
25. P. Brinch Hansen, "Structured Multiprogramming," Commun. ACM, *15*, No 7 (1972).
26. P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," Commun. ACM, *21*, No. 11 (1978).