Top-Down, Bottom-Up, and Structured Programming

CARMA L. McCLURE

Abstract—The process of designing and evolving a programming system cannot be characteri.zedas either top-down or bottom-up or any other well-structured process. The result, however, is a well'structured program.' Since both approaches have the same goal (i.e.; to produce. a highly structured program composed of well-integrated pieces) and follow the same basic philos9phy (i.e., that program composition proceeds inwell thOUght-out systematic steps), either approach or a combination of the two (which is usually what happens) is compatible with a structured programming phiiosophy.

Index Terms-*Bottom-up* development, inside-out approach, levels of abstration, structuredprogramming, top-down development.

INTRODUCTION

THE TOP-DOWN approach is often included as an inherent part of the structuredprograming concept. In the last few years, top-down programming has come into vogue (at least as a buzz word) and appears to be gaining popularity and respect as a useful .designjpro-gramming methodology.. It is not unusual in programming circles to hear that the top-down approach is superior to the bottom-up approach and that structured programming implies a top-down approach. But, what exactly is top-down· programming? What is bottom-up programming? What is their relationship to structured programming? There is confusion concerning the answers to these questions as well as the definition of structured programming itself. An attempt to clarify the concept of structured programming and to determine its influence on the programming process should be accompanied by an examination of the top-down and bottom-up approac.hes and by an investigation of the proposed necessity of· a top-down programming philosophy for a structured approach.

TOP-DOWN APPROACH

There have been attempts to define the top-down approach by a single statement. For example, according to Mills,

> ... top-down programming is a sequence of decomposition and partitions of functional specifications and sub-specifications until finally the level of programming language statements is reached [1].

Randell states

> The top down approach involves starting at the outside limits of the system and gradually working down, at each stage at-teniptingto define what a given component should do before getting involved in decisions as to how the component should provide this function [2].

Manuscript received August 5, 1975.
The author is with the Illinois Institute of Technology, Chicago Ill. 60616.

Since such statements give only a general idea of what top-down programming entails, the steps of a top-down programming approach as explained by Wirth, Dijkstra, and Dahl are outlined in detail below.

Steps oj the Top-Down Approach

Step 1: Begin with an examination of th,e problem, and then propose a "plan of attack," which is given as the general structure of a solution. This solution is developed from a functional viewpoint (i.e., it is expressed in terms of a task which when performed will accomplish the desired objective [3J). In a second point of view (that of Dijkstra [5J), the solution is conceived as a dedicated virtual machine complete with the necessary data structures, instruction set, and a program which can solve the problem. When the program is executed on this machine, the desired task is accomplished. The third point of view (Dahl [4J) expresses the problem solution in terms of a concept (e.g., procedure, hlock) from the programming language which can be implemented in several ways.

Steps 2 *thru n:* Next, in successive steps "refine" the proposed solution by defining its composition in terms .of subtasks..The refinement process is repeated until the task can be directly expressed in terms of a particular programming language [3]. This process can be viewed as two processes, as follows.

1) *An Analysis Process:* The aspects of the solution are examined in greater and greater detail in discrete sequential steps.

2) *A Synthesis Process:* The solution program is built in discrete sequential steps which parallel the. analysis process.

Dijkstra's version of the refinement process is described in terms of machines or pearls. At each step,a machine (pearl) is built. !tis a refinement (definition) of the. entities from which the previous, more abstract machine was built. The process is complete when the machine (pearl) is built entirely of entities which are attributes ofa particular programming language [5]. Dahl sees the process as developing a hierarchy of concepts. In sucGessive steps, complex concepts are defined in terms of more simple ones until finally all concepts correspond to the programming language components [4J.

Refinement Characteristics: The characteristics of the refinement process are the following.

1) At each step inthe process, a set of primitives is introduced which contains: a) an instruction set; and/or b) a data structures set.

2) The steps of the refinement process are marked by design decisions (i.e., each time a design decision is made

or a group of design decisions is made, a new step in the process is begun). These decisions involve the program (a further determination of its logical pieces), its data structures, or both. No order in which to make these decisions has been proposed but the following guidelines have been suggested.

a) Decompose decisions as much as possible so as to separate aspects which may initially appear related [3J.

b) Defer those decisions which concern details of data representation as long as possible [3J, [5].

c) Base design decisions upon criteria such as efficiency, storage economy, clarity, and consistency of structure [3].

d) When making design decisions, alternativ.es should be considered. This means that a family of solutions rather than a single solution to the problem should be developed [3J, [5].

e) Attempt to make the easiest decision first [5].

f) Compose the program in minute steps deciding as little as possible with each step [5].

Theoretically, the refinement process can be thought of as a one pass process, but, practically, this is unlikely to be the case. At each step in the refinement, the decision (or set of decisions) which is considered the most appropriate is selected from a set of alternatives. However, during some successive step, a previous decision may be shown not to be the best. Therefore, one must back-up in the process (perhaps even to the top), choose an alternative decision, and then continue [3].

Design Layers: Each step in the refinement process corresponds to a hierarchical level, machine (pearl), or layer of understanding of the program (see Fig. 1). Step 1 produces layer 1,. Step 2 produces layer 2, etc.

A layer is logically composed of a set of concepts which the designer deemed related and addressable at that point in the program composition process. Alternatively, a layer can be viewed as a set of attributes or primitives. An attribute is either a functional component, which performs a subtask of the program, or a data structure. A layer is considered logically complete since its attributes can be combined by means of an algorithm to form a solution to the programming problem.

The relationship between two layers is defined in terms of their attributes (i.e., the primitives used to define the program at layer i are defined in layer i + 1, those at layer i + 1 are defined in layer i + 2, until at layer n, the primitives are defined in terms of the programming language).

The transition from step to step (layer to layer) is marked by a design decision or a set of design decisions regarding the composition of some program attribute(s).

Example oj a Top-Down Approach: Reservations Program

Program Purpose: The purpose of the Reservations Program is to process reservation transactions. More specifically, the program tasks are the following.

1) Read transactions to create Reservations file.

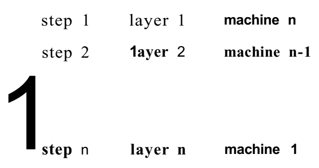

Fig. I. Refinement process.

2) Update Reservations file with new reservations and delete canceled reservations.

3) Print out updated Reservations file.

Refinement Process: At the ith stepin the process Machine (concept) M_i, which can accomplish the tasks of the Reservations Program, is created. The components of M_i are

D_i	data structure set
I_i	instruction set
Ai	algorithm(s) which can be executed on Mi.

An instruction is a functional component of the program. It represents a task to be performed by an algorithm on Mi. An instruction can be decomposed into simpler components (subtasks).

In a top-down approach, the vertical space between the problem solution (program) and the actual programming environment is covered in a downward direction (see Fig. 2). Step 1 produces M_n, which solves the problem in the most generalized terms. Step 2 produces M_{n-1}, and finally Step n produces M_1 which solves the problem in terms of the actual programming environment. The programming environment is reached when all instructions can be expressed either by means of a statement of the programming language or a syntactic category (e.g., SECTION in Cobol or procedure in Algol 60) of the language.

In Algorithm, Ai is expressed in terms of instructions from I_i, data structures from D_i, and instructions and data strUctures assumed available in the actual programming environment. In the notation used here, statements in Ai are terminated by a period, and substatements are terminated by a semicolon. A comment is delimited by an asterisk and a period. The instruction keyword is underlined, and the data structures that it references follow and are enclosed in parentheses.

Step 1: The program is logically divided into three components whose functions correspond to those outlined under Program Purpose.

```
Machine Mn

  Dn(file)
  In(write(file), update(file), print(file))
  An =
      * Create Reservations file.
      write(reservations-file).
      *Update Reservations file.
      update(Reservations-file).
      * Print out updated Reservations file.
      print(Reservations-file).
      stop.
```

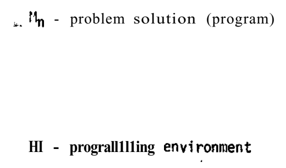

Fig. 2. Top-down approach.

Step 2: Machine M_n is explained in terms of Machine M_{n-1}. An arbitrary design decision must be made concerning the attributes of M_n to be further refined by M_{n-1}. Either the data structure or the instruction set may be refined. In this step, the decision is to expand the notion of updating the file.

Machine M_{n-1}

D_{n-1}(file, record)
I_{n-1}(read(file), write(file), print(file/record)
    find(record), add(record), delete(record))

Definition of a transaction record:
.Flight-number
Date-of-flight
Name-of-passenger-1
Name-of-passenger-2
Transaction-code:
    O-cteate record on new file
    I-update existing file
Update-code:
    O-add reservation to file
    1-delete reservation from file.

Definition of a Reservations file record:
Flight-number
Date-of-flight
Name-of-passenger-I
Name-of-passenger-2

```
An-I:
  * Create Reservations file.
  write(Reservations-file).
  *Update Reservations file.
  while not file-end do
    * Read a transaction.
    read(Transaction-file) at end file-end := 1;
    if not file-end
    - * Find record on Reservation file.
      find(Reservation-record) invalid key in-valid := I;
      * Display message if record is to be updated and
        cannot be found.
      if in-valid
        in-valid := 0;
        print('Invalid record' Reservation-record)
      else if update-code
        * Delete reservation from Reservations file.
        delete(Reservation-record)
      else
      - * Add reservation to Reservations file.
        add(Reservation-record)
    else-
      * Print out updated Reservations file.
      print(Reservation-file).
  stop.
```

Step 3: The notion of adding a record to the file and deleting a record from the file is expanded.

Machine M_{n-2}

D_{n-2}(file, record)
I_{n-2}(write(fiie), read(file), find(record), insert(record),
    print(record), delete(record), modify(record), rewrite
    (record))

```
An-2:
  *Initially create Reservations file.
  write(Reservations-file).
  while not file-end do
    read(Transaction-file) at end file-end: = 1;
    if not file-end
      if not update-code
      * Add a reservation to Reservations file.
      find(Reservations-record) invalid key
        insert(Reservation-record)       --
        print('New record added' Reservations-
          record)
        in-valid := I;
      * Check to see if room on flight for reservation.
      if not in-valid
        if passenger-name-I or passenger-name-2 =
          blanks
          modify(Reservations-record)
          print('Reservation added' Reservations-
            record)
        else
          rewrite(Reservations-record)
          print('No room on flight' Reservations-
            record)
      else
        * Reset invalid record switch.
        in-valid := 0
    else
    - * Delete reservation from Reservations record.
      find(Reservations-record) invalid key
      print('Nonexistent record' Transaction-
        record)
        in-valid := 1;
      * Delete reservation.
      if not in-valid
      - modify(Reservations-record)
        if passenger-name-1 and passenger-name-2 =
          blanks
          delete(Reservations-record)
          print('Record deleted' Reservations-record)
        else
          rewrite(Reservations-record)
          print('Reservation deleted'Transaction-
            record)
      else
        in-valid := 0.
  print(Reservations-file).
  stop.
```

Step 4: The c(;ncept of initially creating the Reservations file is refined.

```
. Machine M_{n-3}

  D_{n-3}(file, record)
  I_{n-3}(report(file), create(file), find(record), insert(record),
    modify(record))
  A_{n-3}:
    while not file-end.
      do read(Transaction-file) at end report(Reserva-
        tions-file) stop;
      if not transaction-code
        * Initially create Reservations file.
        create(reservations-file)
      else
      - * Update existing Reservations file.
        if not update-code
          * Add a reservation to Reservations file.
          find(Reservations-record) invalid key
            insert(Reservations-record)  -
            print('New record added' Reservations-
              record)
            in-valid := 1;
          if not in-valid
            * Check to see if room on flight.
            if passenger-name-1 or passenger-name-2 =
            - blanks  --
              modify(Reservations-record)
              print('Reservations added' Reservations
                record)
            else
              * No room on existing record.
              rewrite(Reservations.-record)
              print('No room on flight' Reservations-
                , record)
          else
          -*-Reset invalid record switch.
            in-valid := 0
        else
        -*-Delete reservation from file.
          find(Reservations-recQrd) invalid key
          print('Nonexistent record'- Transaction-
            record)
            in-valid := 1;
          * Delete reservation from record.
          if not in-valid
            modify(Reservations-record)
            if passenger-name-1 and passenger-
              name-2 = blanks
              delete(Reservations-record)
              print('Record deleted' Reservations-
                record)
            else
             _rewrite(Reservations-record)
              print('R:eservation deleted' Reservations-
                record)
          else
            * Reset invalid record switch.
            in-valid: = O.
```

Program Component Hierarchical Structure

The Program Component Hierarchical Structure is developed as a by-product of the refinement process. It represents the control structure of the program. Entries in the hierarchy are the functional components of the program. At step n in the refinement process each entry in the hierarchy represents an internal procedure or SUBROUTINE (see Fig. 3).

Step 4 is the last step for the Reservations Program. At this point,all the program components can be directly expressed as statements (instructions) from the programming language or by more complex syntactic entities (e.g., SUBROUTINE or SECTION in Cobol or procedure or block in Algol 60) of the programming language. Also, all the data structures can be expressed in terms of the programming language environment. The steps of the top-down approach for the Reservations Program are summarized pictorially in Fig. 4.

BOTTOM-UP APPROACH

Dahl characterizes the Bottom-Up method using his notion of conceptual level:

> A well-formed conceptual level (bottom-up) is a set of well-defined interrelated concepts, which may be combined to make more elaborate concepts.

In the bottom-up case we start at the basic language level and construct abstract concepts capable of capturing a variety of phenomena in some problem area [4].

Steps of the Bottom-Up Approach

Step 1: The bottom-up approach begins with an outline of the general structure of a proposed solution to the programming problem. This outline is expressed in terms of program components (procedural and data).

Steps 2 through n: The concatenation process proceeds in successive steps. This approach can be explained using the layered machine terminology which was introduced for the top-down approach. The bottom-up process also spans the vertical space between the program solution and the programming enviromnent. However, instead of proceeding downward from the virtual solution environment to the programming language environment as the top-down approach dictates, the bottom-up process moves upward from the actual programming environment to a solution (see Fig. 5) [4J, [6].

The general idea is to incrementally rebuild the actual machine M_i which is available into a more suitable machine M_n which can solve the programming problem (see Fig. 6).

Each step in the process is the result of a design decision. The motivation for a design decision in a top-down ap-

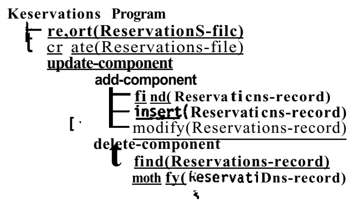
Keservations Program
re,ort(ReservationS-filc)
cr ate(Reservations-file)
update-component
add-component
find(Reservaticns-record)
insert(Reservaticns-record)
modify(Reservations-record)
delete-component
find(Reservations-record)
modify(KeservatiDns-record)

Fig. 3. Hierarchical structure of the Reservations Program.

Top-Down
M4
M3
M2
Mj

Fig. 4. Reservations Program refinement steps.

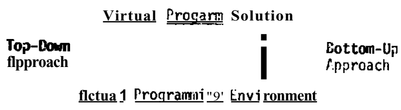

Fig. 5. Top-down and bottom-up processes.

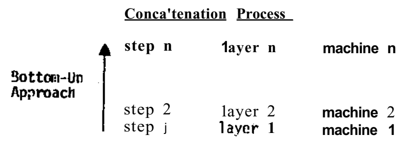

Fig. 6. Bottom-up approach.

proach is functional (i.e., a task is decomposed into subtasks or a data structure is refined into simpler components), but, in a bottom-up approach, it is provisional. At step i, M_i is produced. M_i is an enhancement of M_{i-1} in that it possesses all the attributes of M_{i-1} and then some. M_{i-1} provides a set of primitives from which M_1 can be constructed.

The construction method for program composition in a top-down approach is decomposition (i.e., decompose functions into subfunctions until the program can be defined as a composition of syntactic structures from a prograinming environment). The construction method in bottom-up is concatenation (i.e., combine the programming language attributes into more sophisticated structures which can solve the programming problem).

The bottom-up process stops when a set of program functions and data structures that are able to solve the programming problem have been constructed from the elements available in the actual programming environment.

In practice, the bottom-up'approach, just as the top-down approach, is not a one pass process. It is quite possible to make an incorrect design decision which may not be discovered until later steps. This error could cause a backup in the process to make an alternative decision.

Example of a Bottom-Up Approach: Reservations Program

Programming Problem: The purpose of the Reservations Program is to process reservation transactions. More specifically, the program must do the following.

1) From input transactions, build the Reservations file.

2) Update the Reservations file: a) add new reservations to the file; b) delete canceled reservations from the file.

3) Print the Reservations file.

Step 1: This is the initial formation of the programming approach.' It is done as part of either a top-down or bottom-up approach.

The program structure and general data structure requirements are the following.

1) Data structure requirements:
   a) A file on which to keep reservations.
   b) A file for printing out reservations.
2) General outline of program:
   a) CREATE FILE: Initially create Reservations file.
   b) UPDATE FILE: Update Reservations file.
   c) PRINT FILE: Print out Reservations file.

Step 2: The program composition process is begun. Consider routines which the program will require-in particular, common routines. At each step a design decision is made. Usually, there are several possible decisions from which to choose. For example, some possible decisions to consider at this step are the following.

1) Further definition of Reservations file structure.

2) Definition of various input/output (I/O) routines.

The choice in this example is to define I/O routines. The' following three routines are required by this program.

1) read(file) This routine reads transactions.

2) find(record) This routine finds a particular record on the Reservations file and updates that record.

3) report(file) This routine prints out the Reservations file-.--

Step 3: In order to further define the I/O modules it is necessary to know what information is to be contained in the Reservations file and the type of data structure to represent the Reservations file and Transaction file. Thus, an appropriate decision at this point is to define the Reservations file in greater detail.

1) The information in the Reservations file is:
   Flight-number
   Date-of-flight
   Name-of-passenger-1
   Nanae-of-passenger-2
2) The information in the Transaction file is:
   Flight-number
   Date-of-flight
   Name-of-passenger-1
   Name-of-passenger-2
   Transaction-code:
      1) create record on new file
      2) update record on existing file
   Update-code:
      1) add. reservation to file
      2) delete reservation from file.

Step 4: Again, anyone of several design decisions could be made. However, simply because it seems natural, the decision is to define program components according to their execution time order.

```
CREATE-FILE.
while not file-end :
  read(Transaction-file) at end file-end := 1
  if transaction-code = 1
  . write(Reservations-file).
```

Step 5: The transition from the creation of the Reservations file to its update suggests the next design decision. Create a module which controls the transition from the file creation step to the updatirtg step for the Reservations file.

```
READ-TRANSACTION.
read(Transactions-file) at end file-end := 1.
if not file-end
  if transaction-code = 1
    CREATE-FILE
  else
    UPDATE-FILE
else
  report(Reservations-file).
```

Note that this definition of READ-TRANS forces a redefinition of CREATE-FILE from Step 4.

```
CREATE-FILE.
write(Reservations-record).
```

. *Step* 6: All that remains is to define in detail the modules forupdating a Reservations record.

```
DELETE-RESERVATION.
find(Reservations-record) invalid key
  print ('Nonexistent record' Transaction-record)
  in-valid := 1.
if not in-valid
  modify(Reservations-record) ;
  if passenger-name-l and passenger-name-2 = blanks
    delete(Reservations-record)
  else
    rewrite(Reservations-record) ;
  print('Reservation deleted' Transactions-record)
els-e-
  in-valid := 0.
ADD-RESERVATION.
find(Reservations-record) invalid key
print('Nonexistent record' Transaction record)
  in-valid := 1.
if not in-valid
  if passenger-name-l or passenger-name-2 = blanks
    rewrite(Reservations,.record)
    print('Reservation added' Reservations-record)
  els-e--
    print('No room on flight' Transaction-record)
else — —
  in-valid := 0.
UPDATE-FILE.
if update-code = 1
  DELETE-RESERVATION
else
  ADD-RESERVATION.
```

Step 7: The complete program is as follows:

```
RESERVATIONS-PROGRAM.
read(Transaction-file) at end file-end:= 1.
if not file-end
- if transaction-code = 1
    CREATE-FILE
  else
    * Update Reservations file with transaction.
    if update-code = 1
      DELETE-RESERVATION
    else
      ADD-RESERVATION
else
    report(Reservations-file).
```

Program Component Hierarchical Structure

```
Reservations Program
 └─READ-TRANSACTION
     !-CREATE-FILE
          DELETE-RESERVATION
            ├─find(Reservations-record)
      [     └─modify(Reservations-record)
          ADD-RESERVATION
            Lfind(Reservations-record)
       report(Reservations-file)
```

CONCLUSION

Top-down and bottom-up methodologies are attempts to systematize the program composition process. The result of either approach is a modularized program. How well the parts of the program fit together depends upon the skills of the program designer and his design decisions. The top-d.own (bottom-up) approach is only meant to offer a framework in which the design process can proceed. It cannot give the designer answers to specific design questions for a particular program. It can only guide him (hopefully in the most appropriate design direction) by reminding him that decisions should be made as simply and as explicitly as possible.

For particular problems, one approachmay be preferred over another. According to Randell,

> The top-down approach is for the designer who has faith in his ability to estimate the feasibility of constructing a component to match a set of specifications. The opposite approach is for the designer who prefers to estimate the utility of the component that he has decided he can construct [2].

For example, Dijkstra's THE Operating System was primarily designed and tested bottom-up because he was chiefly influenced by the existing hardware on which the system was to run [7]. Also, if a problem is a generalization of another problem which uses previously designed and tested program blocks, the bottom-up approach may prove most useful.

On the other hand, an argument against the bottom-up approach is that decisions made in the lowest layer impact the whole design. Thus, the use of a top-down approach in which specific decisions concerning the data structures are delayed may be preferred.

Neither approach really claims that it can be followed rigidly or exclusively throughout the program composition process. When designing essentially top-down, one will always peek below into lower layers using previous experience and knowledge of the program language attributes to influence design decisions. When following a bottom-up approach, the designer always will keep in mind the general control structure and the program objective ultimately to be achieved. But whatever approach is used, the design, is not likely to be successful unless the program composition proceeds in well thought-out steps.

REFERENCES

[1] H. Mills, "Mathematical foundations for structured programming," IBM Corp., FSD Rep. RSC 71-5108, 1971.
[2] B. Randell, "Towards a methodology of computing system design," in *Conf. Rec.,* 1968 *NATO Conf. Software Engineering,* pp. 204-208.
[3] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.,* vol. 14, pp. 221-227, Apr. 1971.
[4] O. Dahl and C. A. R. Hoare, "Hierarchical program structures," in *Structured Programming.* New York: Academic, 1972.
[5] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming.* New York: Academic, 1972.
[6] E. W. Dijkstra, "Complexity controlled by hierarchical ordering of function and variability," in *Conf. Rec.,* 1968 *NATO Conf. Software Engineering,* pp. 181-185.
[7J ——, "The structure of 'THE'-multiprogramming system," *Commun. Ass. Comput. Mach.,* vol. n. pp. 341-346, May 1968.

Carma L. McClure received the B.S. degree in mathematics from Loyola University, Chicago, Ill., in 1968 and the M.S. degree in computer science from the Illinois Institute of Technology, Chicago, in 1972. She is currently completing her dissertation for the Ph.D. degree in computer science from the Illinois Institute of Technology.

She is presently a Computer Consultant for Time, Inc., Chicago, Ill., where she is conducting a structured programming in-house training course. Her research interests include the program design process and program testing.

Optimal Placement of Software Monitors Aiding Systematic Testing

C. V. RAMAMOORTHY, MEMBER, IEEE, K. H. KIM, MEMBER, IEEE, AND W. T. CHEN

Abstract-The usefulness of software monitors in testing large 'programs is discussed. Several types of testing strategies based on the use of monitors are surveyed. Since there is a computational overhead involved in employing monitors, attempts are made to minimize the number of monitors employed. This optimization problem is formulated and analyzed in graph-theoretic terms. Implementation aspects are considered through both discussion and examples.

Index Terms-Optimal placement, overhead, program testing, software monitor, test-input, test-output, test-path, test-run.

I. INTRODUCTION

WITH increasing sizes of large real-time systems, the reliability of their software has become a very frequent and serious problem [1]. Software errors can be very disastrous in real-time systems like those of air traffic control and ballistic missile defense [14]. *Complete validation,* which assures the absolute correctness of a program through verification of its complete behavioral characteristics, still remains to be infeasible with any sizable program [2]. Naturally, pragmatic approaches aiming at *partial validation* with high cost-effectiveness via testing have been frequent in practice [3J-[5J, [13].

Program testing is the most common and basic technique aiming at partial validation [6]. A typical testing process involves four major subprocesses: test-path generation, test-input generation, test-run, and test-output evaluation. In order to be cost-effective and still be effective in handling large programs, any validation process should be amenable to a high degree of automation. In spite of its frequent usage and important role, program testing has remained to be more of an art and its complete automation is a distant goal yet. In this paper, we discuss techniques of using software monitors as a partial aid to the systematic performance of program testing. A *software*

Manuscript received August 5, 1975; revised September 15, 1975. This work was sponsored by the National Science Foundation under Grant GJ-35833 and by the Office of Naval Research under Contract NOOOI4-69-A-020Q-I064.

C. V. Ramamoorthy and W. T. Chen are with the Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Calif. 94720.
K. H. Kim is with the Department of, Electrical Engineering, Systems and Computer Science Program, University of Southern California, Los Angeles, Calif.