Structured Programming in Cobol: An Approach for Application Programmers

Allen Van Gelder

Techniques for designing and writing Cobol programs are presented. Previous work in structured programming is drawn upon and adapted. The presentation is informal: the terminology is nonmathematical as far as possible, no theorems are proved, and examples are used frequently. Top-down program design is implemented through the use of structured flowcharts, disciplined specifications, and step by step verification. A well-formed Cobol program is defined. The proper use of the GO TO and other Cobol coding practices are discussed.

Key Words and Phrases: structured programming, top-down, well-formed program, GO TO statement, repeat statement, flowchart, application programming, Cobol, software reliability, program verification

CR Categories: 4.0, 4.22, 5.24

Author's address: 2230 Geary Boulevard, San Francisco, CA 94115.

Introduction

Structured programming has been extensively investigated in recent years as a means of improving the process of software development. Some of its objectives are to reduce system development time and cost, to improve reliability, to produce systems that can be adapted to changing requirements, and to make programming a pleasanter activity. The reports on practical experience with structured programming are favorable [2, 11, 18, 21–24].

Although Cobol is the most widely used language for application programming, it has been the poor stepchild of structured programming: most early work proposed new languages or used more sophisticated languages [24]. However, with the growing popularity of structured programming, Cobol has been receiving its share of attention more recently [11, 20], and an education effort in structured Cobol is underway [21, 24].

The theoretical results in structured programming are now sufficient to develop practical techniques for use in the application programming environment. To transfer this research into practice, and use Cobol in its present form, we must look behind the questions of language and concentrate on what activities and mental processes go into structured programming.

What is structured programming, anyway? There is a widespread misconception that equates it with not using GO TOs. But the process begins long before coding. C.A.R. Hoare, as quoted in [10], has called it "the systematic use of abstraction to control a means of documentation which aids program design." Perhaps the later sections of this paper will help to illuminate this statement. They attempt to draw upon several areas of structured programming and present a unified technique by which specifications can be translated into a Cobol program.

Relation to Other Work

Two major topics of investigation in structured programming are modularization and control flow structures. While both are concerned with software quality, modularization is concerned with dividing up the tasks of a large system, and control flow is concerned with developing procedures to accomplish those tasks. In the former area, Parnas has developed the important concept of "information hiding" in the design and specification of system modules [16, 17].

The top-down design techniques discussed in this paper are mainly applicable below the system module level. That is, after specifications for system modules have been arrived at, by Parnas' or other methods, they would be coded using control flow techniques that promote flexibility, reliability, efficiency, clarity, etc. How this can be accomplished is the problem addressed in the remainder of this paper.

There has been considerable investigation into the question of what is a suitable set of control structures to use in programming. Dijkstra, in a famous note [4], pointed out that arbitrary GO TOs create structures for which proof of correctness is very difficult. It had already been shown that, at the cost of introducing auxiliary variables, any program could be described using only three control structures: succession, if-then-else, and simple (i.e. single-entry, single-exit) loops [3]. (Auxiliary variables are introduced to "remember" results of IFs and loops. They represent processing overhead in that they are set and later tested to achieve flow control that could have been accomplished with GO TOs from inside the IF or loop.) A programming discipline was developed using only these structures [5, 14], and was used successfully to develop a large system [2]. It is possible to implement this discipline in Cobol without use of the GO TO. This approach was used successfully to develop a Cobol system, except that GO TOs were used for certain limited purposes [11].

More recent investigations have raised questions about the desirability of so limited a set of structures [1, 10, 18]. In order to guarantee that auxiliary variables are not needed, it is necessary to include multiple-level exits; however, multiple-entry loops are not required [18]. BLISS, a programming language with no GO TO, contains a WHILE statement to implement single-exit loops, but also an "escape" capability was added to permit multiple-exits and multiple-level exits [22, 23]. The language XPL has a GO TO, a WHILE for iteration, and a means for multiple-level exits. In actual use [13], the GO TO was used only once in about 5000 statements.

Peterson, Kasami, and Tokura defined well-formed programs to be those in which all loops and IFs are properly nested and can be entered only at their beginning, gave a corresponding definition for well-formed flowcharts, and obtained the following results [18]. Any flowchart can be transformed into a well-formed flowchart without introducing any overhead (such as auxiliary variables) into its execution sequence. Any well-formed flowchart can in turn be coded as a well-formed program. They give algorithms for both transformations. Independently, Hecht and Ullman defined reducible flowgraphs to be those that could be reduced to a single node by repeated application of two local transformations [6, 7]. One transformation "absorbs" a succession or if-then-else into the node above it and the other "absorbs" a simple loop into the node it both leaves and enters. They prove that a flowgraph is reducible if and only if it has no multiple-entry loops; therefore the class of well-formed flowcharts coincides with the class of reducible flowgraphs. Furthermore, any well-formed flowchart can be derived in a top-down, stepwise manner by applying the inverses of the two reduction transformations. The above results provide strong arguments for adopting *well-formed control structures* (i.e. those consistent with well-formed programs) as being the set of permissible programming control structures.

There has been much attention given to the possibilities of a language with no GO TO. While this has much theoretical appeal, the practical aspects must also be given careful consideration. Knuth has made a case against its too hasty removal [10]. We are just now beginning to understand what programming languages are all about; in another ten years, he predicts, we will have a really good one. In the meantime, Cobol is the language most widely used in business data processing. Methods for structured programming in Cobol are needed now. The methods presented in this paper make no attempt to eliminate the logical use of GO TOs. It is felt that most abuses of the GO TO result from a lack of proper planning. It is more productive to encourage good planning than to cover up the symptoms of poor planning.

Structured Flowcharts

A new flowcharting method for structured programming will be described now, and used throughout this article. This technique is a modification of the original conception of Nassi and Shneiderman [15]. Some of the advantages of this method are that no arrows are used, no off-page connectors are used, and top-down structure is exhibited. We call flowcharts of the type described in this section *structured flowcharts*.

A *process* is defined as a single-entry, single-exit procedure. The process is the basic building block of both structured programming and structured flowcharts. A process is represented by a rectangle in the flowchart. Assignment statements (MOVE, COMPUTE), procedure calls (CALL, PERFORM), I/O statements, and combinations of these constitute the simplest processes.

The development of a structured flowchart proceeds by incorporating a valid structure into a rectangle already in the flowchart. The valid structures are *succession*, *decision*, and *iteration*. Figure 1 shows the symbols used and illustrates the step by step development of PROCESS-A. As this example shows, the starting point is a rectangle to represent the entire process. This is subdivided using the basic symbols for process, decision, and iteration. New rectangles represent new processes, and may be subdivided in

Fig. 1. Basic flowchart symbols (top row), the paths of control that they imply (second row), and an example that combines them (bottom).

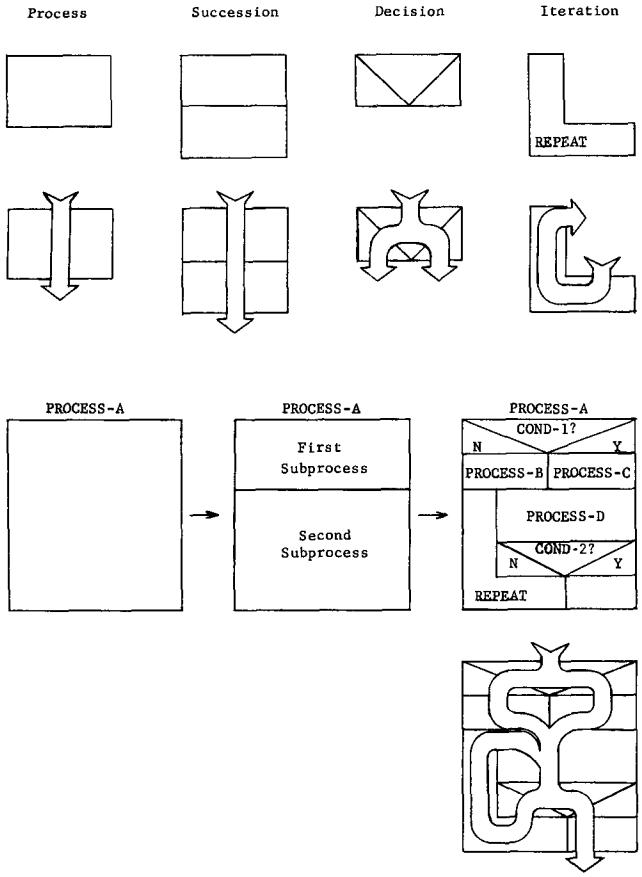
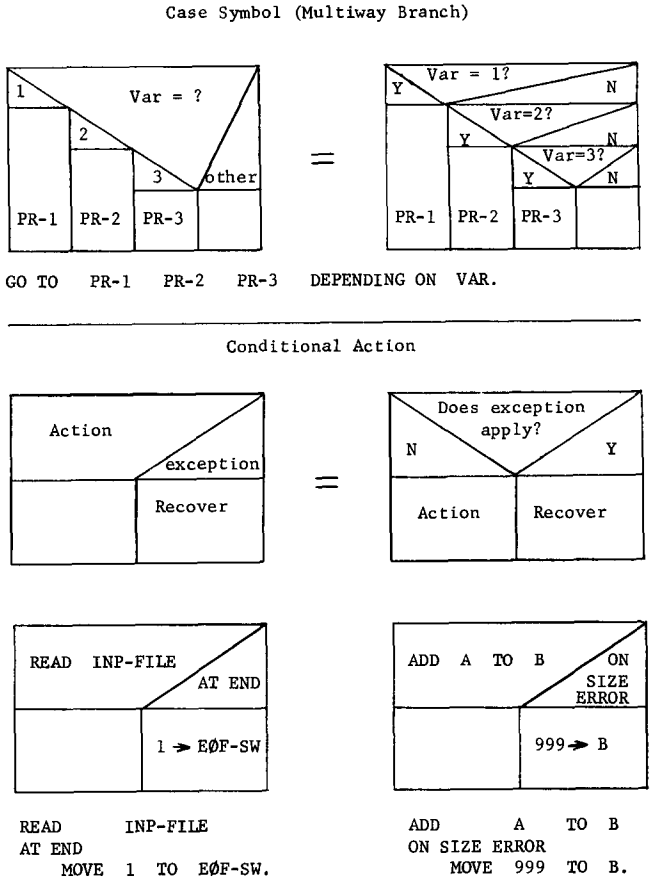


Fig. 2. Variations of the basic decision symbol have equivalents as shown. In the conditional action, note that "Action" does not take place if the exception condition is true.



turn. Any rectangle may have a description of its process written within it.

In a complex process, each flowchart diagram should be kept reasonably simple. To this end, after several subdivisions, names are written *without underlining* in rectangles requiring further subdivisions and a separate diagram for each is drawn. For example, on the left side of Figure 3, the process PRINT-CHAR has a separate diagram to spell out its details. By convention, when the name appears without underlining it is a *reference* to some other diagram or description; when the name is underlined, or is above the diagram, it is a *label.* Circular references, including references to a diagram from within itself are not valid. A flowchart consists of the original diagram and all the related diagrams. These should be ordered so that each reference is to a later diagram. Referenced processes are normally incorporated at their point of reference (i.e. coded in-line) unless the reference is qualified by PERFORM or CALL.

The flow of control implied by the flowchart symbols is illustrated in Figure 1. Since each diagram represents a process, it has an implicit exit at its bottom line where all paths of control (except those from iteration symbols) merge.

The succession structure is drawn simply by dividing the rectangle into two rectangles, one over the other. In Figure 1, this was the first refinement of PROCESS-A.

The basic decision structure consists of the decision symbol with a process (rectangle) under each half of the V. In Figure 1, this refinement was applied to the top half of PROCESS-A, producing COND-1, PROCESS-B, and PROCESS-C.

While the decision symbol is sufficient for all branching of control, it is sometimes convenient to use variations that are equivalent, because they correspond to Cobol syntax. Figure 2 illustrates two such variations, the case symbol, or multiway branch, and the conditional action symbol.

It frequently happens that the identical process appears several times in a diagram that contains several decisions. Figure 3 shows an example of this. As shown there, when identical processes are side by side and are succeeded by the same flowchart element, they may be combined into one rectangle. (Processes at the bottom of a diagram have the implicit exit as their successor.)

There are numerous ways to compose iteration structures. All three basic symbols—process, decision,

and iteration—are used. This refinement was applied to the bottom half of PROCESS-A in Figure 1. There the iteration symbol specifies that if COND-2 is not true, control goes back to PROCESS-D. A number of valid iteration structures are shown in Figure 4.

An iteration structure is valid only if each iteration symbol identifies a single-entry loop. That is, every path of control from the beginning of the diagram to the lower part of an iteration symbol must pass through the process or decision to which that iteration symbol transfers control. Furthermore, there must be a possible exit for each iteration symbol. That is, its lower part must be under some decision symbol whose right half avoids iteration and whose left half leads to iteration. The iteration symbol represents the main departure from [15], in that it allows for multiple-level exits. Note that the multiple exits from *loops* must all eventually merge into a single exit from the *process*.

A well-formed flowchart is defined as one in which all loops are single-entry [18]. It is clear that any structured flowchart has an equivalent conventional flowchart that satisfies this requirement. However, there are well-formed conventional flowcharts that cannot be transformed into structured flowcharts.

The class of iterative structures that can be represented by a structured flowchart represents a compromise. On the one hand, it is known that single-exit loops are sufficient to compose any program, but at the expense of introducing auxiliary variables, which must be redundantly tested to find out what happened in the loop [3, 22]. On the other hand, it is known that any program can be represented by a well-formed flowchart without introducing *any* extra processing (disregarding GO TOs) into its execution [18]. Most practical applications can be formulated using structured flowcharts (a subset of well-formed flowcharts), and no extra processing or auxiliary variables. In unusual cases, auxiliary variables are needed. However, among several examples presented by Knuth [10] that could not be handled conveniently by control structures associated with most go-to-less languages, all but one could be represented by structured flowcharts without introducing auxiliary variables. One, a binary tree insertion problem, required one auxiliary variable.

Process Specifications

The decomposition procedure discussed in later sections depends closely on a certain approach to developing specifications for processes. The decomposition starts with specifications for the process to be decomposed. As it proceeds, new, subsidiary processes, to be invoked by the existing process, are defined. The specifications for the new process must be developed in a coherent manner.

For the purposes of the decomposition procedure, the specifications of a process have four parts: *name*, *initial assertions*, *objectives*, and *data involved in statements of initial assertions and objectives*.

The *initial assertions* are conditions that must hold at the start of the process being specified in order for it to work properly, but which are not checked by this process. Responsibility for their truth is discussed later as part of verification. Initial assertions are normally combinations of data relationships. Simple examples are "A is numeric," "I = 0", and "FILE-1 is open."

The *objectives* in a specification describe how data is to be transformed between the beginning of the process and its termination. In mathematical terms, a process is a function, or operator. The objectives state what the effect of this function should be for all possible configurations of data. Often, an imperative statement serves this purpose, such as "Add 1 to I." A decision table is useful for stating more complex objectives. When used for this purpose, the "actions" in the lower half of the table should only specify changes to be made in the data, and not procedures. The "conditions" in the upper half of the table should be stated in terms of data values as of the start of the process being specified. Intermediate values and work fields should not appear.

The *data* that enters into specifications includes everthing that can affect the operation of the process

Fig. 3. Two occurrences of the identical process PRINT-CHAR can be combined because they are adjacent and have the same successor (in this case, the implicit exit). This process is based on Example 4 from [10].

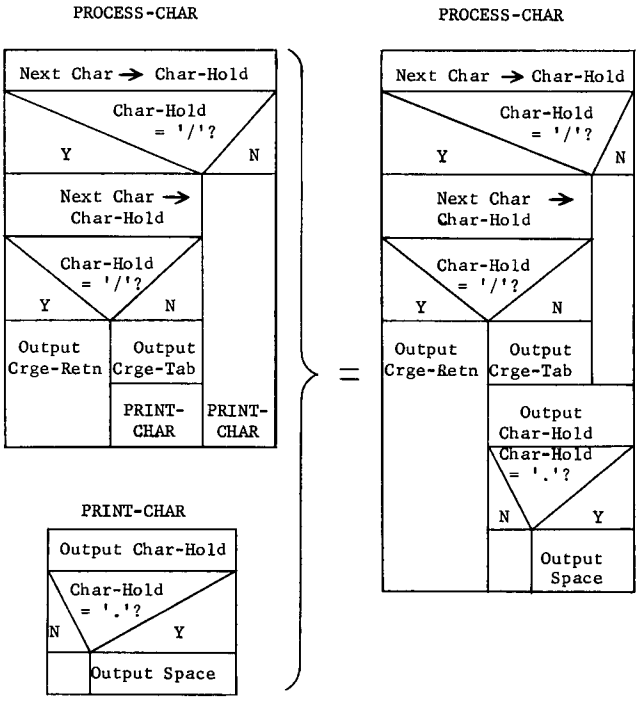
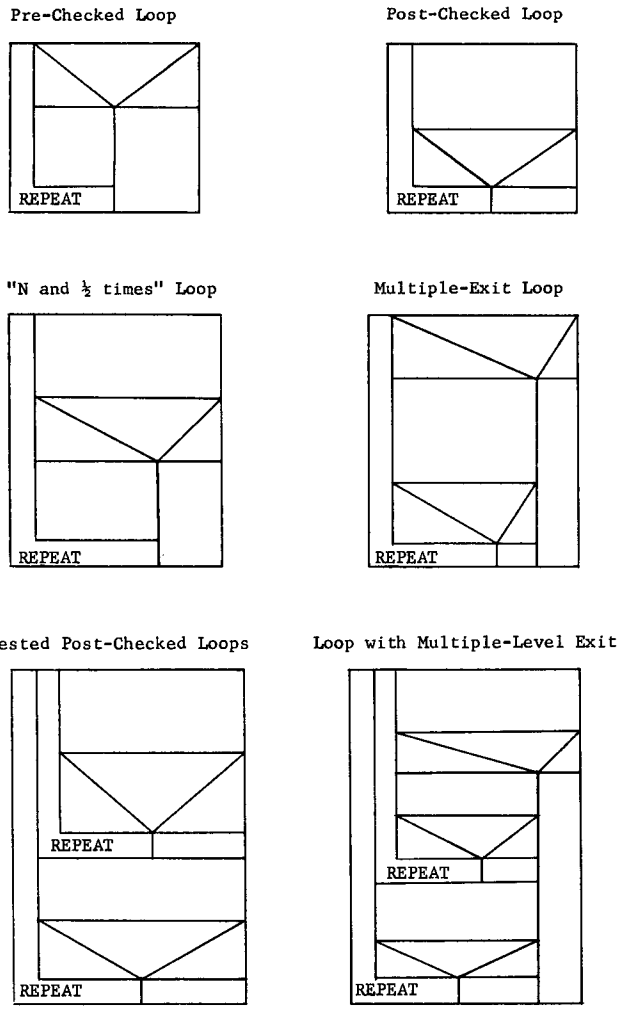


Fig. 4. Several types of loops.



(e.g. through the *conditions* of a decision table), or be affected by the process (e.g. through the *actions* of a decision table). The term "data" is used in a quite general sense in this discussion, and includes external files as well as internal storage. Every program action can be described as an operaton on data of some kind. For example OPEN and CLOSE affect some element of data not accessible to the program, while AT END tests an implicit end-of-file indicator.

Data can be divided into three categories: input, output, and work. These classifications are relative to the process being specified: what is input to one process may be output of another, etc. *Input data* is data not changed by the process. Such input data elements are parameters if they can influence the result of the process; otherwise, they appear only in the initial assertions. *Output data* is data subject to change by the process in order for it to accomplish its objectives. Output data elements are also parameters if their values at the beginning of the process can influence the outcome. Both input and output data must have correct values, in accordance with the specifications, at the termination of the process. On the other hand, *work data* is data used only within the process and does not appear in the specifications at all. Its values are unpredictable at the termination of the process.

There are two shorthand methods that are very useful in writing specifications. The first uses an arrow for assignment operations. The general form is:

expression → data-name-1, data-name-2, . . .

which means: "Evaluate *expression* and store the result in *data-name-1*, *data-name-2*, ..." Typical examples, together with equivalent Cobol statements are listed below.

A → B, C	MOVE A TO B, C
(A + 1) * B → C	COMPUTE C = (A + 1) * B

The second convention distinguishes values of a data-name at the beginning of a process from those at the termination by enclosing the data-name in slashes (/) to denote the value at the beginning of the process. Here are some examples, together with equivalent Cobol statements:

/I/ + 1 → I	ADD 1 TO I
max (/A/, B) → A	IF B > A, MOVE B TO A.

Note that only data-names subject to change need to be enclosed in slashes. (In some technical literature, the "old value" is enclosed in single quotes rather than slashes; however, in Cobol this might be confusing because single quotes are often used to delimit literals.)

HIPO (hierarchy plus input-process-output) has been advanced as a design aid and documentation technique [8]. It could be adapted to the methods of this section by two minor, but important modifications. First, a provision for explicit statement of initial assertions should be added. Second, the *processing* part of the HIPO diagram should be limited to objectives, as described in this section; the "how to," including mention of lower-level processes, is better indicated with a structured flowchart.

Decomposition Procedure

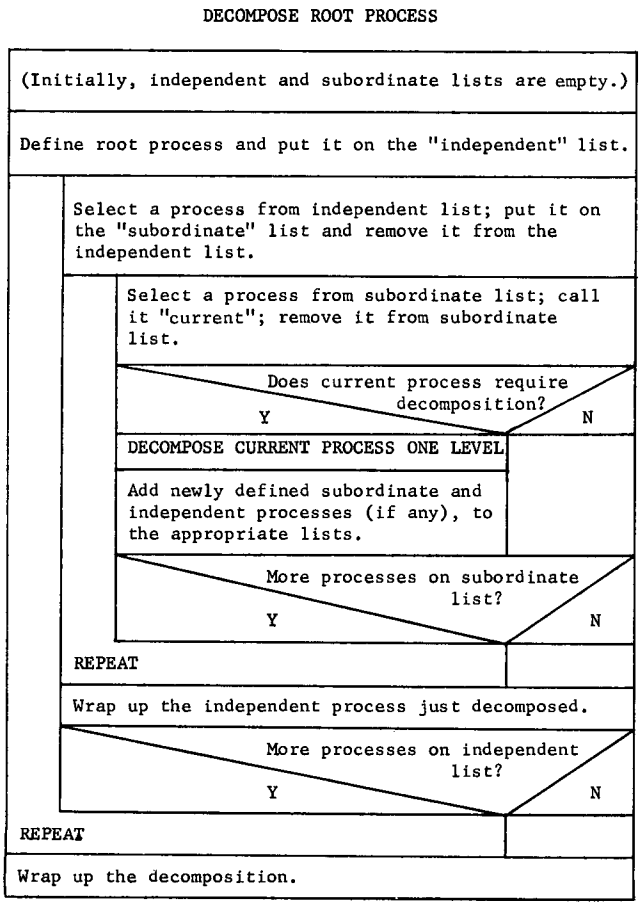
A methodical procedure to decompose a complex process into a set of simpler processes related by flowchart diagrams is discussed in this section. For convenience, the discussion is limited to processes to be coded in a Cobol program; however, it will be obvious that the general method has other applications. The process being decomposed may correspond to one complete program, but need not. The objectives of the decomposition are to express the complex process in terms of flowchart diagrams (of the type presented earlier) and simple processes, and to organize them in such a way that the required program can be coded methodically and will have desirable structural prop-

erties. Here a *simple* process is one that can be coded or understood without further analysis.

The starting point for the decomposition procedure is a single process, called the *root process*, whose specifications are given. During decomposition, new, less complex processes are defined and specified; these are called *subordinate processes* or *subprocesses* if they are unique and relevant only to their immediate parent; they are called *independent processes* if they occur multiply or perform a complete and separate function of general application. Independent processes must be incorporated into the root process by means of PERFORM or CALL statements. Typical examples of independent processes are date conversion, file handling, and standardized routines. The terms root and subordinate are relative, and depend on the scope of process under study at the time. If a root process has subprocesses, then any subprocesses of the subprocesses are also considered to be subprocesses of the root. Clearly, a root process, together with all its subprocesses, forms the structure called a tree. However, any independent processes that were defined during the decomposition of the root process are not part of that tree, but instead are the roots of their own trees. At the bottom level are the subprocesses that are sufficiently simple not to require decomposition; these correspond to the leaves of the tree.

A flowchart for the complete decomposition procedure is shown in Figures 5–7. Essentially, we specify all subprocesses of the root, as well as any needed independent processes. We collect and organize this work, then start over with any unresolved independent process as the root. As we go along, we may decide that several subprocesses should be redefined as one independent process. Finally, we collect and organize the various independent processes (including the root process), and we are ready for coding. The details of this procedure are discussed in the following sections.

Fig. 5. Flowchart for decomposition procedure.



One-Level Decomposition

Given a process that is specified in terms of its initial assertions and objectives, let us see what is required to decompose it one level. (We shall call the process to be decomposed the *current process*. Also, to avoid excessive verbiage, "subprocess" will include independent processes where no confusion is likely.)

The first step is to use a structured flowchart diagram to represent the current process in terms of subprocesses. In many cases a fairly obvious breakdown suggests itself, but sometimes this step requires imagination and creativity to visualize one or more simpler processes that can be combined to produce the desired current process. One level of decomposition should be expressible in a fairly simple flowchart diagram.

Each subprocess in the diagram must be defined in terms of its own initial assertions and objectives. The data involved in each definition should be noted, as it represents the interface between the current process and the subprocess. This data should be categorized as input, output, or work, relative to the current process. Input and output data will appear in the specifications of the current process, while work data will not. However, work data will normally appear in the specifications of the subprocesses.

The next step is to *verify* that the flowchart diagram and the subprocesses actually produce the current process. The word "prove" is avoided because the process of formal proof is too time consuming to be practical in most cases. Verification often consists only of a mental check to ensure that the original thought is sound. The methods discussed below have a rigorous foundation [5, 14].

It is important to remember that the verification task applies only to the single-level decomposition just performed. Therefore we are entitled to *assume* the following:

A1. The initial assertions of the current process hold.
A2. Each subprocess will accomplish its objectives, provided its own initial assertions hold when it is entered.

Fig. 6. Details of one-level decomposition.

DECOMPOSE CURRENT PROCESS ONE LEVEL

Construct flowchart to represent current process in terms of subprocesses (and/or independent processes).
For all newly created processes: SPECIFY PROCESS
Note data interfaces between current process and each subprocess (or independent process).
VERIFY ONE-LEVEL DECOMPOSITION
Whenever appropriate, define an independent process to replace several subprocesses, adjust data interfaces, and recheck verifications. (Some of the replaced subprocesses may have been defined during previous decompositions.)
For each data element: either bind it to the current process or pass it to one subprocess (or independent process).

Fig. 7. Details of process specification and one-level verification.

SPECIFY PROCESS

Assign the process a name.
State its initial assertions.
State its objectives.
Note what data is involved in the above statements.

VERIFY ONE-LEVEL DECOMPOSITION

Assume initial assertions of current process hold.
Assume that subprocesses will achieve their objectives if their initial assertions hold upon entry from current process.
Verify that initial assertions of subprocesses hold upon entry from current process.
Verify that objectives of current process are achieved.

Our task is to verify two things:

V1. The initial assertions of each subprocess hold when it is entered.

V2. The objectives of the current process are accomplished when the bottom of the diagram is reached.

Figures 8–11 illustrate the verification requirements for the elementary decompositions: succession, decision, and iteration. Since any flowchart is built up out of these structures, its verification requirements are the logical extension of the requirements illustrated.

The verification requirements for the succession and decision decompositions are straightforward. In spite of this, considerable writing is necessary to set down all details of even the simplest examples.

It is simply not practical to write 10 or 20 lines of support for each line of code. Programmers are advised to write several examples out completely in order to learn and thoroughly understand what is involved in verification. Then they should make it a mental process. Initial assertions and objectives should still be written for the more important processes, but may be only mentally noted for the numerous intermediate processes that occur in level by level decomposition.

The verification requirements for decompositions involving iteration are the most difficult. First of all, there is a multiplicity of configurations possible, including single-exit loops, multiple-exit loops, and multiple-level-exit loops. Only the single-exit loop is discussed here. The extension to multiple-exit loops is straightforward, and the ambitious reader can develop an extension to multiple-level-exit loops. There are two key steps in verifying an iterative decomposition: the flowchart extension transformation, and the generalized initial assertions.

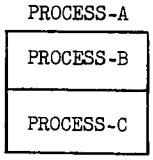
The flowchart extension transformation consists of constructing a flowchart equivalent to the one containing the iteration symbol. It is identical except that the entire process is substituted for the iteration symbol, as shown in Figure 10. The transformed flowchart is a combination of succession and decision. The verification rules for those decompositions can be applied. In accordance with those rules, we are entitled to assume that if the original process is reentered *with the initial assertions intact*, that it will achieve its objectives. This assumption is not entirely warranted because there is a possibility that an iterative process will enter an endless loop and never terminate. Therefore, an additional step is needed to verify that (V3) the process must eventually terminate.

Since an iterative process begins not just once, but every time the REPEAT is reached, its initial assertions must be stated with sufficient generality to be true at every beginning. Arriving at a proper statement of the initial assertions for even a simple iterative process can be a difficult task. In Figure 11, the third initial assertion of PROCESS-A illustrates the type of generalized statement that is often needed. One way to arrive at the required statements is to ask what conditions of data will ensure both that the objectives of the iteration are achieved if it decides to terminate on this cycle, and that the iteration can successfully enter another cycle if it does not terminate.

In view of the difficulty of verifying the correctness of even a simple iteration, it is logical to expect this to be an area where many errors occur in practice. Experience seems to bear this out. Therefore, the prudent programmer will keep his iterative processes as simple and clear as possible.

The last step in the one-level decomposition concerns locality of data. In general, it is desirable to make

Fig. 8. Verification example for the *succession* decomposition. Rules A1, V1, etc. are stated in the text.



PROCESS-A
Objective: (Highest multiple of 4 that is not greater than X) → Y.
Initial Assertion: X is numeric and non-negative.

PROCESS-B
Objective: Set Z to the integer for which: $\frac{X}{4} - 1 \leq Z < \frac{X}{4}$.

Initial Assertion: X is numeric and non-negative.

PROCESS-C
Objective: 4 * Z → Y.
Initial Assertion: Z is numeric.

Verification:

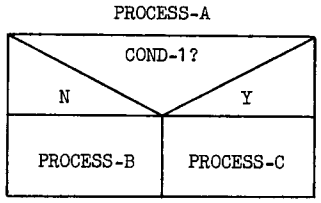
PROCESS-B V1: Initial assertions hold by A1, because they are identical to those for PROCESS-A. By A2, objective is achieved.

PROCESS-C V1: Initial assertions hold because PROCESS-B makes Z numeric. By A2, objective is achieved.

V2: Combining PROCESS-B and PROCESS-C makes Y a multiple of 4 such that Y ≤ X and (Y + 4) > X, as required.

```
COBOL coding:  DIVIDE    X  BY  4  GIVING  Z
               MULTIPLY  Z  BY  4  GIVING  Y.
```

Fig. 9. Verification example for the *decision* decomposition. Rules A1, V1, etc. are stated in the text.



PROCESS-A

Objective: If X is numeric, add it to Y.

Initial Assertion: /Y/ is numeric.

COND-1: X numeric?

PROCESS-B
Objective: No change.
Initial Assertion: None.

PROCESS-C
Objective: /Y/ + X → Y.
Initial Assertions: X numeric; /Y/ numeric.

Verification:

If COND-1 is false:
V1: Initial assertions for PROCESS-B hold, vacuously.

V2: Doing nothing achieves the objective of PROCESS-A.

If COND-1 is true:
V1: /Y/ is also numeric (by A1), so initial assertions for PROCESS-C hold.
V2: /Y/ + X → Y achieves the objective of PROCESS-A.

```
COBOL coding:  IF  X  NUMERIC
               ADD  X  TO  Y.
```

data as localized as possible, that is, to restrict its use to the lowest possible subprocess. The purpose of this step is to determine what data can be made more local than the current process, and to associate with the current process that data that has reached its reasonable limit of locality.

All data that appears in the specifications of the subprocesses into which the current process has been decomposed represent the interface between the current process and its subprocesses. Only subprocesses that require decomposition are considered. Each data element not already associated with a higher-level process must be passed on to a subprocess if possible (and reasonable), or else associated with the current process.

The above considerations lead to the following rules. If a data element is used by more than one subprocess, or no subprocess, it is associated with the current process, as it has reached its limit of locality there. Also, if the current process is standardized and the subprocess is not, the data element is associated with the current process; standardized processes should be accompanied by their own standardized data. In most other cases, the data is passed on to the one subprocess that uses it, and its association is postponed until that subprocess is decomposed. However, when the amount of data that would be passed on is small and the processes are at a fairly low level, it is more practical to associate all data with the current process than to pass it on and get excessive fragmentation. (See the later section on arrangement of Working-Storage.)

After all required one-level decompositions have been done, the processes and data fields are arranged in preparation for coding, as discussed in the following sections.

Arrangement of Processes and Working-Storage

When all one-level decompositions are finished, the processes should be arranged in the order in which they are to appear in the program. As mentioned before, each independent root process has its own subordinate processes, which form a tree under it. The various independent trees are arranged in a sequence such that all PERFORMs refer to independent processes later in the sequence than the process containing the PERFORM.

Within an independent process, all subprocesses have a natural sequence implied by the tree structure, called *depth-first spanning order*. To achieve it, start at the root of the tree and arrange the subtrees one level under it in top-bottom order if one succeeds the other, or in left-right order if they are on the two sides of a decision structure. Thus in the cases of Figures 8, 9, and 10 the sequence would be: PROCESS-A, PROCESS-B and its subprocesses, PROCESS-C and its subprocesses. Repeat this sequencing operation in all

subtrees until all leaves (lowest level processes) have been arranged. (There is no reason not to keep the processes in this order all the way through the decomposition procedure and avoid the need to rearrange.)

Working-Storage fields (and other entries in the Environment and Data Divisions) should be grouped generally according to the processes that use them. During decomposition each field was associated with one process. Now they should be arranged in the same order as those processes. Data fields associated with one process may be further arranged into input data, output data, and work data. During the coding it is very helpful to include comments indicating with which process the various blocks of data are associated, and whether they function as input, output, or work.

Well-Formed Cobol Programs

The term "well-formed Cobol program" is used to describe a program in which loops and conditional sentences are properly nested and entered only at their beginning. The term is formal, and does not automatically imply good quality. The Procedure Division (except for DECLARATIVES SECTIONs) must be a series of nonoverlapping "processes," only the first of which contains a program terminating statement (STOP RUN, EXIT PROGRAM, etc.). There must be no GO TOs from one process to another. Further, each PERFORM in one process must refer to another process that appears later in the program. Internally, each process must satisfy the rules of this section, which are based on the general definition [18], as adapted to Cobol.

An *upward GO TO segment* is any piece of program code that begins with the statement immediately *after* a given procedure-name and ends with a (later) GO TO that references that name. (It may contain other, *embedded* procedure-names.) In Figure 12 an upward go-to segment begins with "IF BA9-I ..." and ends with "GO TO-B110-FIND-LOOP."

In a *well-formed Cobol program*, the following rules must hold:

P1. An upward GO TO in a conditional sentence (containing IF, AT END, ON SIZE ERROR. etc.) must be the last statement of the sentence.

P2. All GO TOs that refer to a procedure-name *embedded* in an upward GO TO segment must be in that same upward GO TO segment.

P3. Multiple-valued GO TOs (i.e. those with the DEPENDING ON option, or subject to change by ALTER statements) must refer only to procedure-names that appear later in the program. Rule P2 must be satisfied for all possible values of such a GO TO.

Code that implements a structured flowchart follows these rules almost automatically.

Fig. 10. Flowchart extension transformation for a typical *iteration* decomposition.

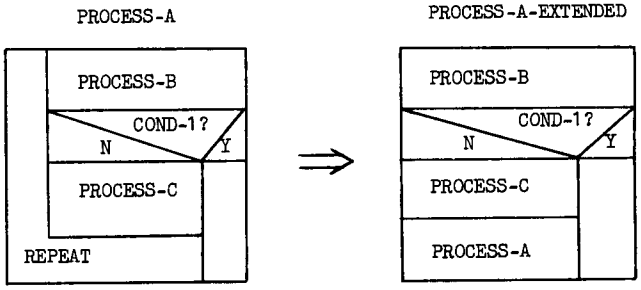


Fig. 11. Example of verification for the *iteration* decomposition shown in Figure 10. Rules A1, V1, etc. are stated in the text.

PROCESS-A
Objective: Set J so that X (J) is the maximum of X (1) thru X (N).

Initial Assertions:
(a) N ≥ 2.
(b) I and J are numeric and between 1 and N, inclusive.
(c) X (J) is the maximum of X (1) thru X (I - 1). (The iteration may start with J = 1 and I = 2.)

PROCESS-B
Objective: If X (I) > X (J), I → J.
Initial Assertion: I and J are numeric and between 1 and N, inclusive.

COND-1: I ≥ N?

PROCESS-C
Objective: /I/ + 1 → I.
Initial Assertion: /I/ is numeric.

Verification (Refer to PROCESS-A-EXTENDED.):

PROCESS-B V1: Initial assertion holds by A1 and (b). By A2, objective is achieved. Also true is:
V* After PROCESS-B completes, X (J) is the maximum of X (1) thru X (I).
Reason: Combine objective of PROCESS-B and assertion (c).

If COND-1 is false:

PROCESS-C V1: I is numeric. By A2, objective is achieved.

PROCESS-A V1: By A2 for PROCESS-C, /I/ = I - 1; then by V*, assertion (c) holds; (a) and (b) are seen to hold also.
V2: By A2, PROCESS-A objective is achieved (provided it terminates).

If COND-1 true:

V2: By (b), I = N; hence V* is the same as the objective of PROCESS-A.

V3: I increases by 1 each time PROCESS-C executes. Other processes do not affect I or N. Therefore, COND-1 is true eventually, and PROCESS-A ends.

COBOL coding:

```
        MOVE  2  TO  I
        MOVE  1  TO  J.
PROCESS-A-LOOP.
    IF  X (I)  GREATER  X (J)
        MOVE  I  TO  J.
    IF  I  LESS  N
        ADD  1  TO  I
        GO TO   PROCESS-A-LOOP.
```

It is theoretically possible to code a Cobol program without any GO TOs, using only the three basic structures: succession, decision, and pre-checked single-exit iteration (via PERFORM with the UNTIL option). However, such a rigid approach can lead to awkward situations, as pointed out by numerous authors in a recent symposium [20]: actions that require more than one sentence for correct syntax cannot be coded in-line as one alternative of an IF ... ELSE; with PERFORM ... UNTIL, the body of the loop cannot be coded in-line; and Cobol has no effective "case" capability for multiple, mutually exclusive alternatives. Some authors offered suggestions for future language changes to overcome these difficulties. Some, concentrating on use of Cobol as it is today, have presented methods that deal with these drawbacks in various ways, including limited GO TO usage [11, 12, 19, 21]. Others are less specific, but argue that the GO TO should not be banned from present day programming [1, 9, 10]. All agree that GO TO usage should be severely curtailed: it may be used to get around *language* inadequacies, but not *structural* inadequacies. The methods presented in this section have one simple, but flexible rule: the GO TO may be used when needed to implement a diagram of a structured flowchart.

Each "level of abstraction" of a process, normally represented by one diagram of a structured flowchart, should be coded as a separate unit, or segment. Its data items should be placed together in the Data Division, in one or more 01 level entries, as needed. The procedure statements should normally run to about 50 lines (one page) or less. A reference within the diagram to a subprocess is programmed as a PERFORM with the THRU clause, except that short, simple subprocesses near the bottom levels of the program might be coded in-line. A diagram reference to an *independent* process is programmed as a PERFORM *without* the THRU clause, or as a CALL to a separate subprogram. To fit in with this convention for PERFORMs, the top level of an independent process is coded as one complete Cobol SECTION; immediately following it (if required) is *one* additional SECTION, containing *all* subprocesses of this independent process. This is illustrated in Figure 12, along with other procedure naming conventions. In the example, the number of levels is exaggerated, of course.

Paragraph names are formed according to certain conventions, in order to exhibit program structure. They contain a prefix (suggested: letter, then three digits), and a descriptive part. Names at the beginning of loops always end with LOOP. Paragaph names that mark the *end*, rather than the beginning of a decision, case or iteration structure consist of a prefix only. Each subprocess is terminated by an EXIT paragraph, whose name is in the form "prefix-EXIT." For readability, subprocesses should be separated by a comment line of asterisks; EJECT may also be used to start the compiler listing on a new page.

The prefixes, although confusing upon cursory examination, indicate the structural relationships among subprocesses. The range of prefixes in each subprocess is nested within the range of its parent process in such a way that, if the subprocess were brought in-line, its prefixes would fall in sequence with the parent's. Thus a programmer can easily find his way around an unfamiliar program. Note also that the prefixes of the first paragraphs of all subprocesses are in ascending order, as a consequence of arranging the subprocesses in depth-first order.

Conclusion

A top-down program design technique has been presented. It emphasizes the importance of organizing

Fig. 12. An independent process coded as root and three subprocesses.

```
 WORKING-STORAGE  SECTION.
 01  BA-B000-AREA.
     05  BA9-CNT        OCCURS 100  PIC S9(03).
     05  BA9-I                      PIC S9(03).

 01  BB-B100-AREA.
     05  BBX-TEST-KEY               PIC  X(04).
     05  BB9-NUM-KEYS   VALUE ZERO  PIC S9(03).
     05  BBX-KEY        OCCURS 100  PIC  X(04).

 PROCEDURE  DIVISION.
 ....

* * * * * * * * * * * * * * * * * * * * * * * * * *
 B000-COUNT-KEYS  SECTION.
 B010-INITIALIZE.
     MOVE  1  TO  BA9-I.
     PERFORM  B100-FIND-MATCH  THRU  B199-EXIT
     PERFORM  B200-UPDATE-CNT  THRU  B299-EXIT.
 B999-EXIT.  EXIT.

* * * * * * * * * * * * * * * * * * * * * * * * * *
 B000-SUBPROCESS  SECTION.
 B100-FIND-MATCH.
*DOC SETS BA9-I SO THAT TEST-KEY = KEY (BA9-I),
*DOC ADDING TEST-KEY TO TABLE IF NOT FOUND.
 B110-FIND-LOOP.
     IF  BA9-I  GREATER  BB9-NUM-KEYS
         GO TO  B119.
     IF  BBX-KEY (BA9-I)  =  BBX-TEST-KEY
         GO TO  B199-EXIT.
     ADD  1 TO  BA9-I
     GO TO  B110-FIND-LOOP.
 B119.
     PERFORM  B150-ADD-KEY  THRU  B159-EXIT.
 B199-EXIT.  EXIT.

* * * * * * * * * * * * * * * * * * * * * * * * * *
 B150-ADD-KEY.
*DOC PUTS TEST-KEY IN AN AVAILABLE SLOT, ZEROS
*DOC THAT CNT, SETS BA9-I, ADJUSTS NUM-KEYS.
     ADD  1 TO  BB9-NUM-KEYS
     MOVE  BB9-NUM-KEYS  TO  BA9-I
     MOVE  BBX-TEST-KEY  TO  BBX-KEY (BA9-I)
     MOVE  ZERO          TO  BA9-CNT (BA9-I).
 B159-EXIT.  EXIT.

* * * * * * * * * * * * * * * * * * * * * * * * * *
 B200-UPDATE-CNT.
     ADD  1 TO  BA9-CNT (BA9-I).
 B299-EXIT.  EXIT.
* * * * END OF  B000-SUBPROCESS  SECTION  * * * *
```

the program into processes that operate at varying levels of abstraction, such that each one has manageable complexity. The program can then be coded naturally into a well-formed Cobol program. GO TOs should be used during coding as the need arises.

Received May 1975; revised August 1976

References

1. Abrahams, P. "Structured programming" considered harmful. SIGPLAN Notices (ACM) *10*, 4 (April 1975), 13–24.
2. Baker, F.T. System quality through structured programming. Proc. AFIPS 1972 FJCC, Vol. 41, Pt. I. AFIPS Press, Montvale, N.J., pp. 339–343.
3. Bohm, C., and Jacopini, G. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM 9*, 5 (May 1966), 366–371.
4. Dijkstra, E.W. Go to statement considered harmful. Letter to the editor, *Comm. ACM 11*, 3 (March 1968), 147–148.
5. Dijkstra, E.W. Notes on structured programming. EWD249, Technische Hogeschool Eindhoven, Eindhoven, Netherlands, Aug. 1969.
6. Hecht, M.S., and Ullman, J.D. Flow graph reducibility. *SIAM J. Computing 1*, 2 (June 1972), 188–202.
7. Hecht, M.S., and Ullman, J.D. Characterization of reducible flow graphs. *J. ACM 21*, 3 (July 1974), 367–375.
8. HIPO—a design aid and documentation technique. GC20-1851-1, IBM, White Plains, N.Y., May 1975.
9. Jackson, M.A. Designing and coding program structures. Symp. on Structured Programming in COBOL—Future and Present, April 1975, pp. 22–53.
10. Knuth, D.E. Structured programming with go to statements. *ACM Computing Surveys 6*, 4 (Dec. 1974), 261–301.
11. McClure, C.L. Structured programming in COBOL. SIGPLAN Notices (ACM) *10*, 4 (April 1975), 25–33.
12. McConnell, J. Developing structured programs using the COBOL language as it exists today. Symp. on Structured Programming in COBOL—Future and Present, April 1975, pp. 177–209.
13. McKeeman, W.M., Horning, J.J., and Wortman, D.B. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, N.J., 1970.
14. Mills, H.D. Mathematical foundations for structured programming. FSC 72-6012, IBM Fed. Systems Div., Gaithersburg, Md., Feb. 1972.
15. Nassi, I., and Shneiderman, B. Flowchart techniques for structured programming. SIGPLAN Notices (ACM) *8*, 8 (Aug. 1973), 12–26.
16. Parnas, D.L. A technique for software module specification with examples. *Comm. ACM 15*, 5 (May 1972), 330–336.
17. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Comm. ACM 15*, 12 (Dec. 1972), 1053–1058.
18. Peterson, W.W., Kasami, T., and Tokura, N. On the capabilities of while, repeat, and exit statements. *Comm. ACM 16*, 8 (Aug. 1973), 503–512.
19. Ryge, S. Structured programming without changing COBOL. Symp. on Structured Programming in COBOL—Future and Present, April 1975, pp. 226–243.
20. Stevenson, H.P., Ed. Symp. on Structured Programming in COBOL—Future and Present, Los Angeles, Calif., April 1975.
21. Weiland, R.J. Experiments in structured COBOL. Symp. on Structured Programming in COBOL—Future and Present, April 1975, 210–224.
22. Wulf, W.A. Programming without the GOTO. Information Processing 71, North-Holland Pub. Co., Amsterdam, 1971, pp. 408–413.
23. Wulf, W.A. A case against the GOTO. Proc. ACM Annual Conf., Vol. II, Aug. 1972, pp. 791–797.
24. Yourdon, E. Teaching structured COBOL to the masses. Symp. on Structured Programming in COBOL—Future and Present, April 1975, 115–133.